MASTER IN COMPUTER SCIENCE

The modeling and analysis of concurrent processes using Petri nets

Horst, Wilmes

Award date:
1985

Awarding institution:
University of Namur

INSTITUT D'INFORMATIQUE
FNDP NAMUR

THE MODELING AND ANALYSIS OF CONCURRENT PROCESSES USING PETRI NETS

MEMOIRE PRESENTE PAR
HORST WILMES
EN VUE DE L'OBTENTION
DU DIPLOME DE
LICENCIE ET MAITRE
EN INFORMATIQUE

ANNEE ACADEMIQUE
1984/85

Acknowledgments

First of all I would like to thank Mr. Ramaekers for having accepted to conduct this thesis.

I am grateful to Mr. Engelhardt and Mr. Ries whose cooperation lead me to the domain of this thesis.

I would also like to thank all the other people involved in the elaboration of this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

Today's computer systems are structured as a collection of sub-systems called processes, almost always running in parallel. These processes can run on a single CPU and share it dynamically, or be distributed over many processors. Recent advances in hardware technology have made possible the construction of distributed systems and multiprocessors. The parallel execution of processes leads to more powerful and reliable systems. But in order to achieve this goal, the processes must share resources and cooperate. This is done by synchronization and communication between processes.

The advantages of parallelism have as a counterpart the complexity of constructing and debugging of such systems. The non-sequential programming makes it more difficult to find out the correct execution of a program consisting of several processes. This is due to the combinatorial explosion of the number of states of the entire system in function of the number of states of the components on the one hand, and the difficulty or inability of observing the states of the whole system on the other hand. This can sometimes be aggravated by the system being non-deterministic.

It is then indispensable to be able to verify and validate the system in construction. This cannot be done by informal reasoning or testing because these methods are not reliable enough. That is why we need a formal analysis method permitting to model the system and to state expected properties.

Petri nets are abstract formal models for the analysis of concurrent systems. Their inherent parallelism makes them suitable for the representation of systems of concurrent activities. Petri nets have such analytical properties that the behavior of the modeled system can be analyzed in a systematic manner with respect to important properties. The essential advantages over other verification methods (e.g. assertions) is the fact that most of the analytical

methods are mechanizable. Petri nets deal essentially with the control part of computation, which is a fundamental aspect of synchronizing software. If synchronization mechanisms can be formulated by means of Petri nets, the analysis mechanisms of Petri nets can be applied to these synchronization mechanisms.

There are two types of Petri net theory. Pure Petri net theory is the study of Petri nets to develop analytical tools used to verify properties. The second direction is the applied Petri net theory. Applied Petri net theory is the application of the results of pure Petri net theory to systems modeled by Petri nets. The direction followed in this dissertation is rather the second one.

The first chapter introduces the definitions used throughout the following chapters. Chapter two shows how to model systems of concurrent activities and gives an overview of some synchronization and communication mechanisms modeled by Petri nets. The different properties to verify and a method for doing the analysis are introduced in chapter three. Chapter four describes a software tool implementing the analysis method proposed in the preceding chapter and discusses some implementation details. Chapter five concludes this dissertation with the application of the automatic analysis technique to some synchronization problems.

CHAPTER 1: PETRI NET CONCEPTS

In this chapter the Petri net concepts used throughout this dissertation are defined. First of all, the structure of a Petri net and the corresponding graph are described, then the concept of marking is defined, followed by an enumeration of the different firing rules. At the end of this chapter, the concepts of marking class and reachability set are introduced.

1. Petri net structure and graph

A Petri net is a four-tuple PN = (P,T,I,O), with :

1. a finite non-empty set of places $P = \{p_1,..,p_n\}$,
2. a finite non-empty set of transitions $T = \{t_1,..,t_m\}$,
3. a forward incidence function or input function : I:PxT -> N,
4. a backward incidence function or output function : O:PxT -> N.

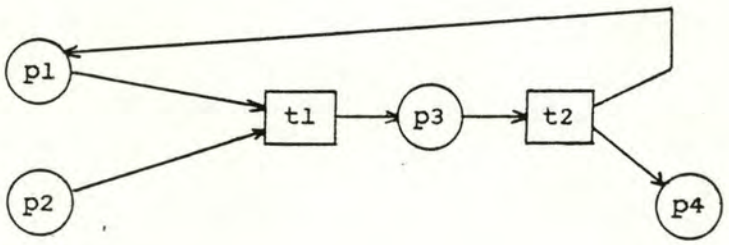
The sets P and T are disjoint ($P \cap T = \emptyset$). In the figures, the places are represented by circles and the transitions by squares or bars.

Two matrices I and O are associated respectively with the input and output functions. Input places of a transition t are such that $I(t,p) > 0$ and an arc labeled by the value of $I(t,p)$ is drawn from place p to transition t (if $I(t,p) = 1$ the label is not required). Output places are such that $O(t,p) > 0$, and are connected to transition t by an arc outgoing from this transition and eventually labeled by the value $O(t,p)$.

From the definition of a Petri net it follows that a Petri net graph is a bipartite graph, i.e. only nodes of different types (T or P) can be linked by arrows.

In Figure 1.1 the places "processor free" and "job awaiting processing" contain the conditions for the succeeding transition "start processing" (these places are input places to the transition).

This transition effects the transfer from the input places to the "job running; processor in use" place (output place of the transition), which is now the condition for the "processing completed" transition. This last transition has two output places : "job completed" and "processor free".



p1 : Processor free
p2 : Job awaiting processing
p3 : Job running; Processor not free
p4 : Job completed

t1 : Start processing
t2 : Processing completed

Figure 1.1 : A simple computer system

2. Markings

A state, called "marking" , of a Petri net is a map $M:P \rightarrow N^n$ with $n = |P|$. A marking may be indicated on the graph by indicating, for every place P, a number M(p) in the corresponding circle or, where M(p) is sufficiently small, putting M(p) dots or tokens in the circle corresponding to p. A place p is called marked if $M(p) \geq 1$.

In Figure 1.1 for instance, we can indicate that the processor is free and that a job awaiting processing is present by putting one token on each of the places representing these conditions (p1 and p2).

As the marking represents the state of the system and this state

can change, the marking of the Petri net must change too. The initial state of the system is represented by an initial marking. The transition from one state to another is defined by the firing rules.

3. Firing rules

The number and the position of tokens in the Petri net changes by the firing of transitions. In order to fire, a transition must be enabled or firable. A transition t is enabled by a marking M if and only if :

$$(\forall\, p \in P) \qquad M(p) \geqslant I(t,p).$$

Notation : $M[t\rangle$

If an enabled transition fires , it changes the marking by removing $I(t,p)$ tokens if p is an input place to t and adding $O(t,p)$ tokens if p is an output place to t. The new marking M' is such that :

$$(\forall\, p \in P)\ M'(p) = M(p) - I(t,p) + O(t,p)$$

We can say that transition t is enabled by marking M and that its firing leads to the marking M', the follower marking of M, and note this :

$$M[t\rangle M'$$

During the execution of a Petri net, (i.e. the firing of transitions) always only one transition can fire at a time even if more transitions are enabled.

Figure 1.2 (a) shows a Petri net with an intial marking. In this situation the transition t1 is enabled since the only input place of this transition is p1 and p1 is marked. Figure 1.2 (b) shows the Petri net after the firing of the transition t1. The only enabled transition is still t1 because a firing of t2 requires two tokens to reside in place p3. This will be the case after having fired t1 a second time. The resulting marking is shown in Figure 1.2 (c). Note that in that situation, the two transitions are enabled.

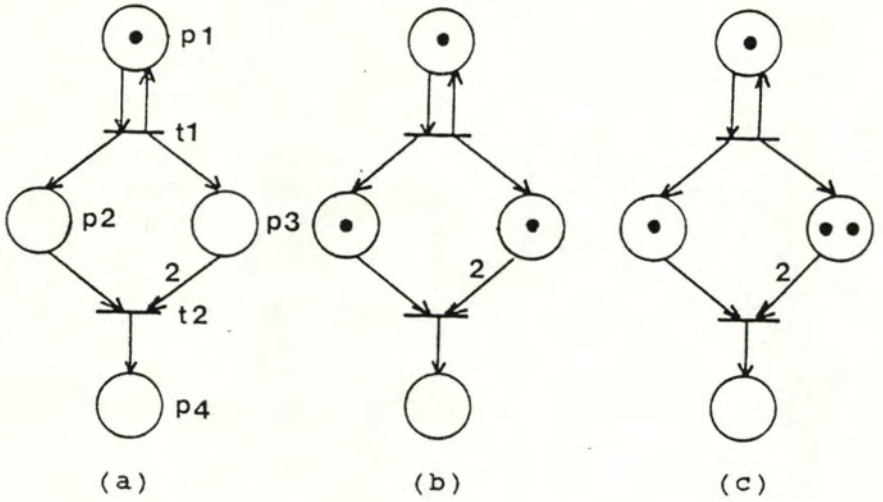


Figure 1.2: The firing of transitions

4. Marking class and reachability set

A marking Mk is reachable from a marking M if

$\exists\ (t_1,..,t_k) \in T$ and $\exists\ (M_1,..,M_{k-1}) \in N^n$ such that

$M[t_1>M_1, M_1[t_2>M_2, .., M_{k-1}[t_k>M_k$

i.e. there exists a firing sequence leading from a marking M to a marking M_k.

Notation : M[->Mk

The set of all markings reachable from a marking M is called the marking class and noted :

$$[M] = \{M\} \cup \{M_i \mid M[->M_i\}$$

The marking class of the initial marking M_0 of a Petri net is called the reachability set of the Petri net and represents the state space of the system modeled by the Petri net.

Figure 1.3 (a) - (c) illustrates the process of firing of transitions of the Petri net presented in Figure 1.1 and how to obtain the marking class or the reachability set.
The initial marking is $M_0 = (1,1,0,0)$. Transition t1 is enabled and fires to give the marking $M_1 = (0,0,1,0)$. Now transition t2 is firable and fires giving the marking $M_2 = (1,0,0,1)$. The execution of the Petri net halts at this marking because no other transition is enabled.

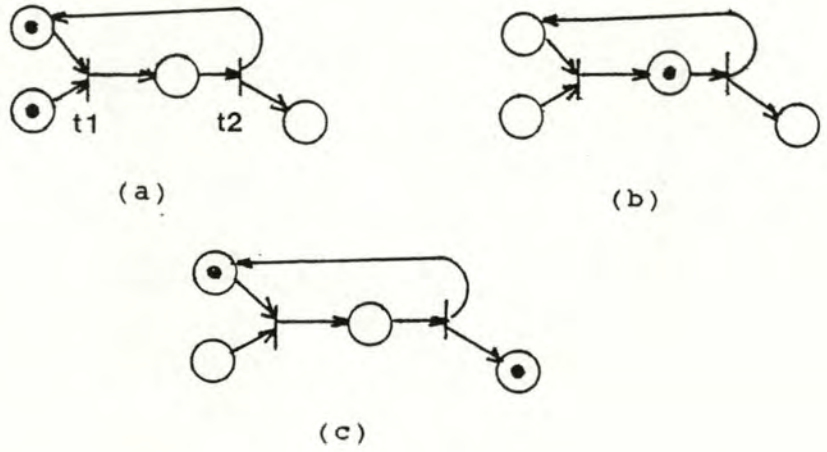


Figure 1.3: Execution of the simple computer system

Thus, we have :

- $M_0[t1>M_1$ (M_1 is a follower marking of M_0),
- $M_0[->M_2$ (M_2 is reachable from M_0),
- $[M_0] = \{M_0, M_1, M_2\}$ is the reachability set and describes all the possible states in which the system can be with respect to the initial state.

CHAPTER 2: MODELING OF CONCURRENT PROCESSES

In this chapter, it will be shown how to model systems of concurrent activities with Petri nets. First some classical synchronization problems will be modeled with Petri nets, then some basic primitives for the expression of concurrency are described.

1. Classical synchronization problems

1.1. Mutual exclusion

The problem of mutual exclusion is defined by two or more processes executing concurrently but each process having a "critical section" which must be executed without any other process executing its own critical section at the same time. Here is how Dijkstra [7] stated the problem : "In considering two sequential processes, "process 1" and "process 2", they can for our purposes be regarded as cyclic. In each cycle a so-called "critical section" occurs, critical in the sense that at any moment at most one of the two processes is allowed to be engaged in its critical section. In order to effectuate this mutual exclusion, the two processes have access to a number of common variables. We postulate that inspecting the present value of such a common variable and assigning a new value to such a common variable are to be regarded as indivisible, non-interfering actions."

A Petri net model representing this problem and a solution to it is given in Figure 2.1.

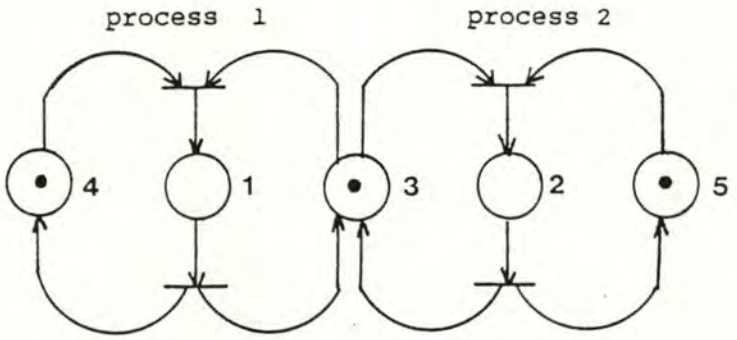


Figure 2.1: The mutual exclusion problem

This model corresponds to the description given by Dijkstra. The two processes are cyclic, each process has a critical section: place 1 for process 1 and place 2 for process 2. Place 3 represents a common variable each process has access to. The other places correspond to different values of the instruction pointers of the two processes. The restriction that the reading of a common variable and assigning a new value are indivisible and non-interfering holds with respect to the definition of the firing rules for Petri nets.

The solution given in this model corresponds to the solution given by Dijkstra using a binary semaphore variable. In chapter 5 it will be shown that the model gives a correct solution to the problem, i.e. it will be shown that :

1. at any moment at most one of the processes is engaged in its critical section;
2. the decision which of the processes is the first to enter its critical section cannot be postponed to eternity;
3. stopping a process in its remainder of cycle has no effect upon the others.

Point two will not be proved since it is an assumption on how a Petri net fires : as long as there are enabled transitions the execution i.e. the firing of transitions will not halt.

This problem of mutual exclusion has been generalised to N cyclic processes, each having a critical section. A solution to this problem

with N=4 is given in Figure 2.2.

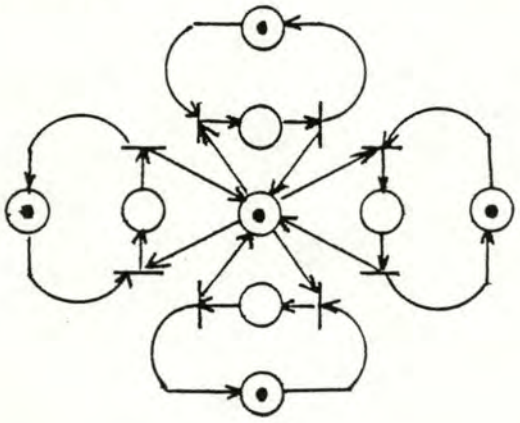


Figure 2.2: The general mutual exclusion problem
(for 4 processes)

1.2. The Dining Philosophers Problem.

This problem was originally stated and solved by Dijkstra [8]. It can be stated as follows : Five philosophers (processes) are at a dinner at a round table; each one alternately eats and thinks. In the middle of the table is a bowl of rice and a chopstick is placed between each adjacent pair of philosophers (5 chopsticks). From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him i.e. philosopher i picks up both chopsticks i and i+1 (where 5+1=1). When a philosopher succeeds in picking up the two chopsticks, he can eat without releasing them. When he has finished, he puts them down and starts meditating again.

There is a solution to this problem (Figure 2.3) which is not correct since deadlock can occur. This solution represents the fact that each philosopher first takes one chopstick and only after having acquired it takes the other one. In this solution it can happen that all philosophers succeed in acquiring one chopstick but no one can take the second because all chopsticks are in the hands of philosophers.

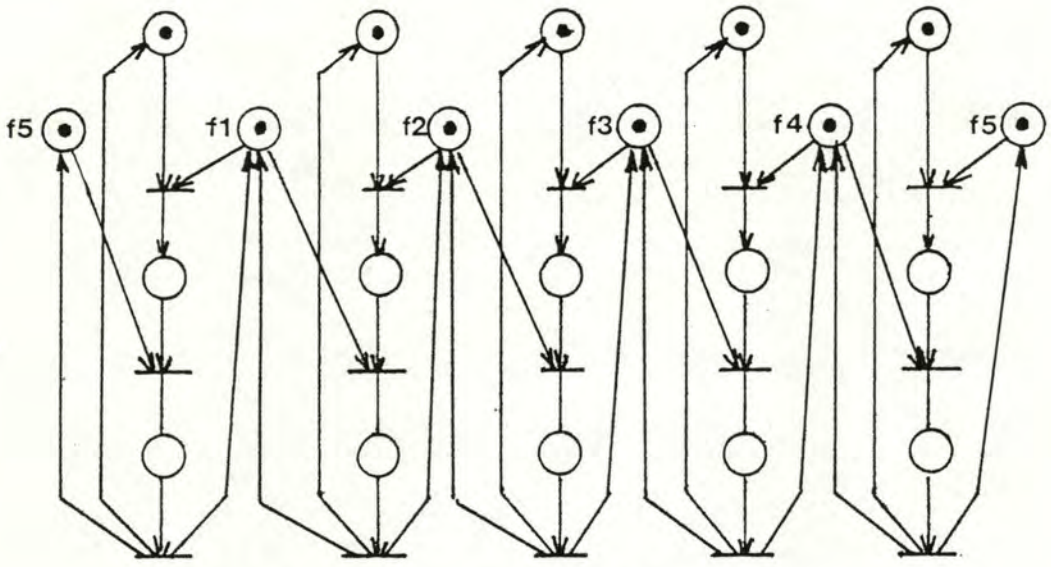


Figure 2.3: Philosophers problem

A deadlock is avoided in the correct solution by a philosopher taking both chopsticks at the same time. This solution is shown in Figure 2.4.

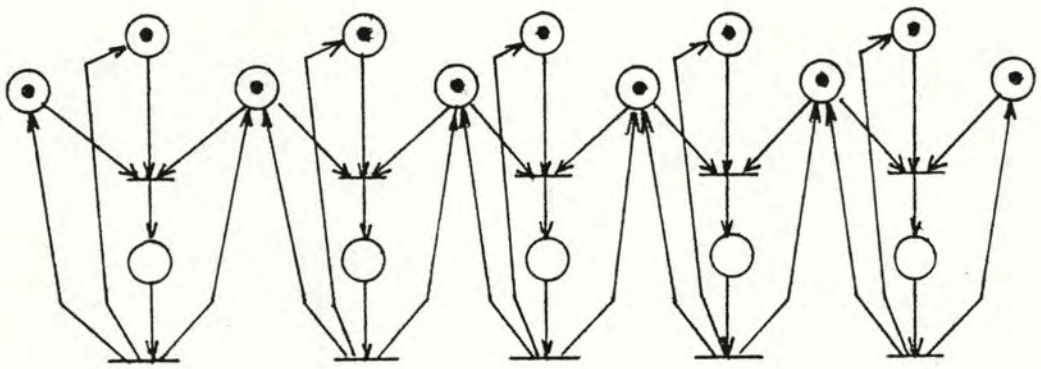


Figure 2.4: Philosophers : a deadlock-free solution

A problem not solved in this last solution is starvation. It is possible that two philosophers could cause starvation of the philosopher sitting between them by alternately eating and preventing the philosopher in the middle to take the two chopsticks needed to eat. A possible solution preventing starvation is presented in Figure 2.5. The philosophers take always one chopstick at a time as in the first solution, but only 4 philosophers are admitted to pick up

chopsticks at a time to prevent a deadlock situation.
In the second solution a philosopher can starve because wanting to eat he must wait until the two chopsticks closest to him are on the table. But it can happen that the chopsticks are never at the same time on the table. The third solution does prevent from starvation because a philosopher always waits for only one chopstick and sooner or later the chopstick will be available if the system cannot run into a deadlock situation as in the first solution. The deadlock prevention in this solution is realized by permitting only to a maximum of four philosophers to pick up chopsticks.

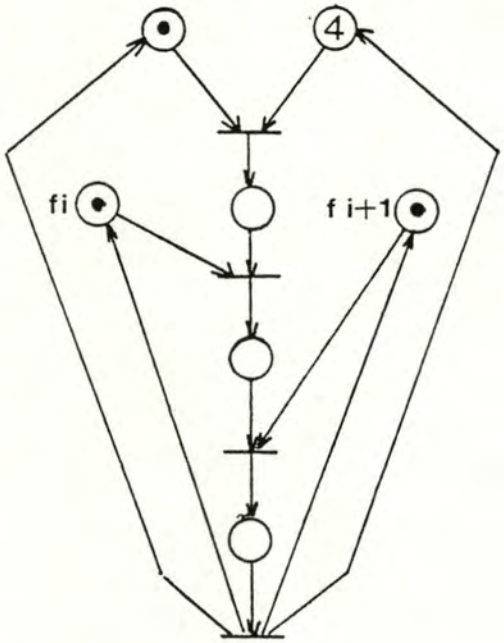


Figure 2.5: Philosophers problem : starvation free

In chapter 5 it will be shown that the first solution is not correct and results in a deadlock situation and that the second and the third one are both correct solutions whith respect to deadlock.

1.3. The Producer/Consumer problem

An important synchronization problem is that of the Producer/Consumer. The producer and consumer are both cyclic processes. In each cycle, the producer produces some information that the consumer has to consume in one of his cycles. Each time the producer produces data, he deposits it in a buffer. The consumer removes data from the buffer to consume it. In the "unbounded buffer" producer/consumer problem, it is assumed that the buffer is of unlimited capacity.

The producer and consumer must be sychronized, so that the consumer does not try to consume items which have not yet been produced. A Petri net representing such a producer/consumer pair is given in Figure 2.6.

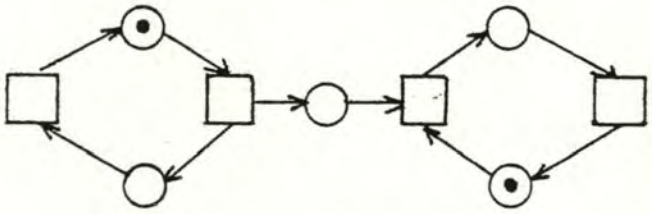


Figure 2.6: Unbounded Producer/Consumer

An unbounded capacity of the buffer is not a realistic assumptions. This leads us to the following problem statement. The "bounded buffer" producer/consumer problem assumes that the buffer has a limited capacity N. In this case, the producer can only produce if the buffer is not full, in order to prevent an overflow of the buffer. The behavior of the consumer can remain the same as for the unbounded buffer problem. The corresponding Petri net is shown in Figure 2.7.

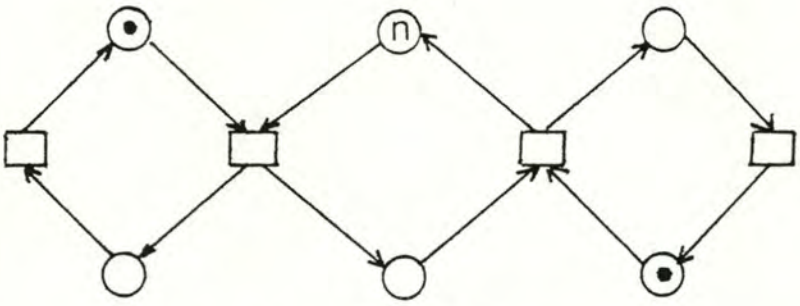


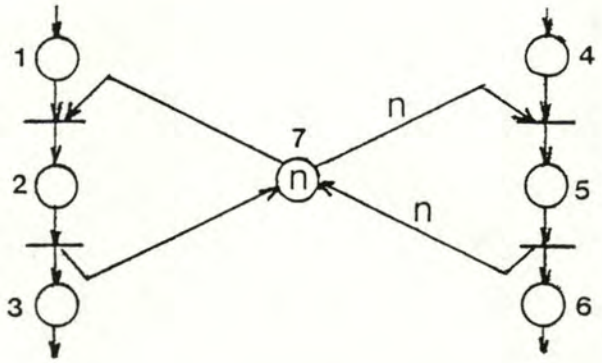
Figure 2.7: Bounded Producer/Consumer

1.4. The Readers/Writers Problem

This problem was stated in [6]. There are two classes of processes sharing a data object. The processes of the first class, named writers, must have exclusive access to the object, but processes of the second class, the readers, may share the object with other readers.

In the bounded version of the "first" readers/writers problem, up to n reader processes may read simultaneously and no reader should be kept waiting unless a writer has already obtained the permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. A Petri net representing this situation is given in Figure 2.8.

The second readers/writers problem is the same as the first one except that if a writer is ready to write, no reader may start reading until all (up to m) waiting writers have finished writing. The Petri net in Figure 2.9 models the second readers/writers problem.



p1 (p4) : process wanting to read (write)
p2 (p5) : process reading (writing)
p3 (p6) : process finished reading (writing)
p7 : synchronizing place

Figure 2.8: Readers/Writers I

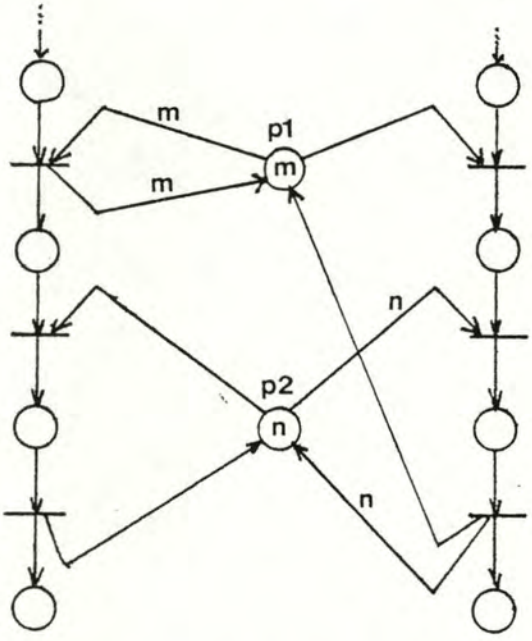


Figure 2.9: Readers/Writers II

A process wanting to read can only start reading if there are m tokens on place p1, i.e. there is no process wanting to write, and if there is at least one token on place p2, i.e. there are not yet n processes reading concurrently. A process wanting to write must first take one token from p1 to indicate to the readers that he wants to write. Then the writer must wait until no process is reading (n tokens on place

p2) and can start writing. Place p1 and p2 represent synchronizing variables and the other places are dwell-points for instruction pointers of processes.

2. Synchronization Primitives

2.1. Parbegin and Parend

When modeling systems of concurrent activities, we must be able to specify that some actions are executed concurrently. For this purpose, Dijkstra [7] introduced extensions to ALGOL 60 to enable someone to describe parallelism of execution :
"When a sequence of statements - separated by semicolons as usual in ALGOL 60 - is surrounded by the special statement bracket pair "parbegin" and "parend" this is to be interpreted as parallel execution of the constituent statements."
The example given by Dijkstra can be modeled by means of Petri nets as shown in Figure 2.10. The formulation of the example is the following :

"begin S1; parbegin S2;S3;S4 parend; S5 end"

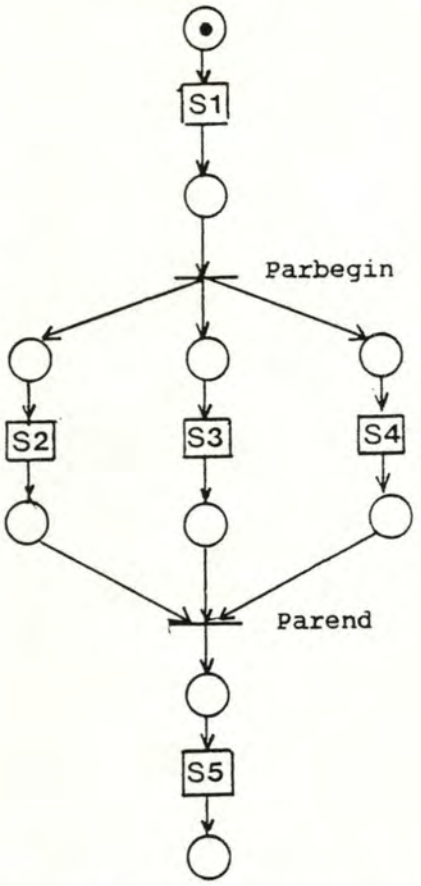


Figure 2.10: Parbegin and Parend

Other primitives with the same functionality have been introduced as for example the cobegin - coend construct. Thus, the modeling primitives for such constructs are a transition with multiple output places for the parbegin and a transition with multiple input places for the parend.

2.2. Fork and Join

With the parbegin - parend construct, the process including the pair of instructions is suspended until all constituent statements are executed and then only resumes the main process.

The fork instruction however produces two concurrent executions, one starting at the label specified by the fork instruction, the other being the continuation of the process emitting the fork. The join

instruction permits to recombine two concurrent computations into one.

The Petri net model for this construct (Fork–Join) is the same as the one for parbegin – parend, but the interpretation differs: the fork starts one new process and executes it concurrently to the process containing the fork instruction whereas the parbegin creates two new processes and suspends the calling process until completion of the processes created by the parbegin.

2.3. The Semaphore

A semaphore is an integer variable whose value can only be altered by the operations P and V defined as follows [7]:

- the P–operation decreases the value of its argument semaphore by 1 as soon as the resulting value would be nonnegative. The completion of the P–operation is to be regarded as an indivisible operation.
- the V–operation increases the value of its argument semaphore by one.

A semaphore which has a maximum value of one is called a binary semaphore; if the maximum value of a semaphore is greater than one, it is called a general semaphore. Figure 2.11 shows how a semaphore and the operations upon it are realized by means of a Petri net.

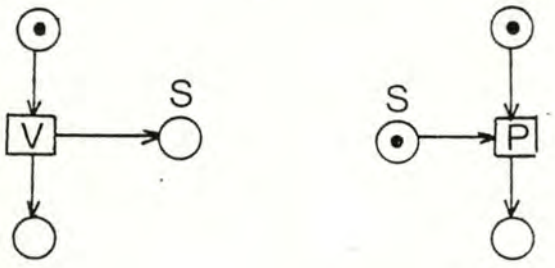


Figure 2.11: P/V

The semaphore is represented by a place S and its initial value by the corresponding number of tokens on that place. The P–operation is a transition taking a token from the semaphore place; the V–operation puts one token on the semaphore place.

Remark : This model is a functional representation of the semaphore

concept; i.e. the semaphore and the operations defined on it are shown as the user of such constructs sees them; the implementation details are hidden.

2.4. Message passing

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. An interprocess communication facility basicallly provides two operations : send and receive. A process executes send to pass a message to another process; the other process accepts information by executing a receive.

When we study message passing systems, we are not interested in the data flow taking place betweeen processes, but in the synchronization, i.e. the control flow of the processes modeled by Petri nets. The property of message passing systems influencing the control flow is the capacity of the link between the processes. The capacity determines the number of messages that can termporarily reside in the link. There are three types of capacities leading to three different models for message passing systems:

1. unbounded capacity : the link between two processes can contain an infinite number of messages. The sender can always continue after executing a send and he is never delayed. This situation corresponds to the unbounded buffer problem and is represented in Figure 2.12.

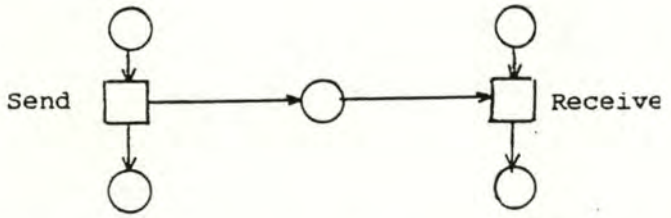


Figure 2.12: Send/Receive (unbounded capacity)

2. bounded capacity : The link is of bounded capacity n; thus at most n messages can reside in it. If the link is not full when a

message is sent, it is placed in the link and the sender can continue without waiting. If the link is full, the sender is delayed until a message is removed from the link by a receive operation. Figure 2.13 shows this construct.

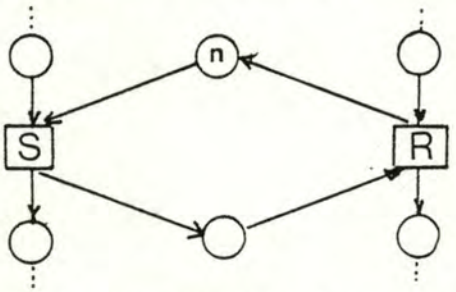


Figure 2.13: Send/Receive (bounded capacity)

3. zero capacity The link has a capacity of zero messages, i.e. no message can be queued. In this case, the two processes must be synchronized for a message transfer to take place. If the send occures first, the sender is blocked until the receive occurs; then the transmission of the messsage takes place and both processes are allowed to proceed. Conversely, if the receive occurs first, the receiver is blocked until the send occurs. This synchronization is also called "rendezvous". In this method, the sender never proceeds before the receiver has effectively received the message. This arrangement is shown in Figure 2.14.

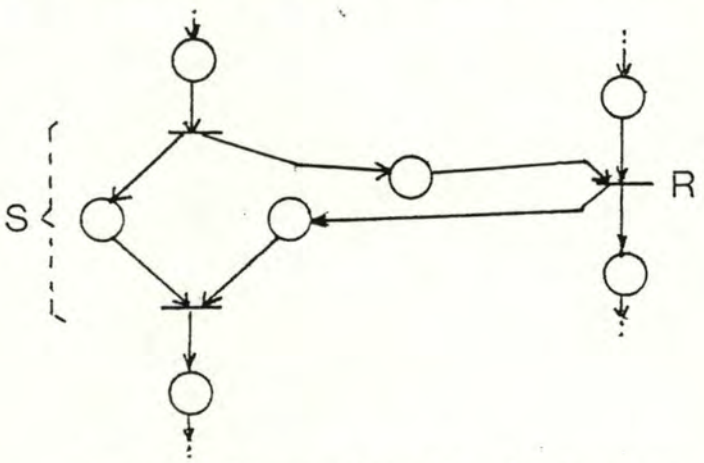


Figure 2.14: Send/Receive (Zero-capacity)

The zero capacity message passing method is implemented in programming languages such as CSP and OCCAM [11,18]. The main features of these languages are :

1. Dijkstra's guarded commands for introducing and controlling nondeterminism,
2. a parallel command, based on Dijkstra's parbegin,
3. input and output commands are used for communication between concurrent processes,
4. no automatic buffering : the communication is synchronized (0-capacity message passing),
5. Input commands may appear in guards to permit a process to wait for input from any one of a number of channels. The input is taken fom the first channel on which output by another process is available.

2.5. The ADA "Rendezvous"

The rendezvous mechanism in ADA is based on the "blocking send" which is an extension of the zero-capacity message passing. In this case, the answer permitting the resumption of the sender is not given by the receive operation but has to be given explicitly. The "blocking send" scheme eliminates send and receive and replaces them by three new operations: BlockingSend, Accept and Reply. Accept can only receive a message sent by BlockingSend, and Reply can only answer a message received by Accept. This makes it possible for the receiver process to perform some action before giving the acknowledgment (Reply) to the sender, Notice that if no action is performed before replying, this scheme is the same as the zero-capacity message passing.

In ADA [17,12], the message and the reply (if any) are parameters. The send is an entry point invocation of the receiving process, the accept is the "accept" statement and the reply corresponds to the "end" of the accept-block (do..end).
The rendezvous thus achieves the following three basic notions [17].

1. Synchronization : The calling task must issue an entry call, and the called task must reach a corresponding accept statement.

2. Exchange of information : at the realization of the rendezvous, parameters can be received by the acceptor. After the end statement, parameters may be passed back to the caller process.
3. Mutual exclusion : If two or more tasks call an entry point of a task, only one call can be accepted at a time.

As in CSP and OCCAM, a guarded command construct is available : the select statement. It provides a task with a mechanism to wait for a set of events whose order cannot be predicted in advance.
The synchronization between an entry point invocation and an accept statement are shown in Figure 2.15.

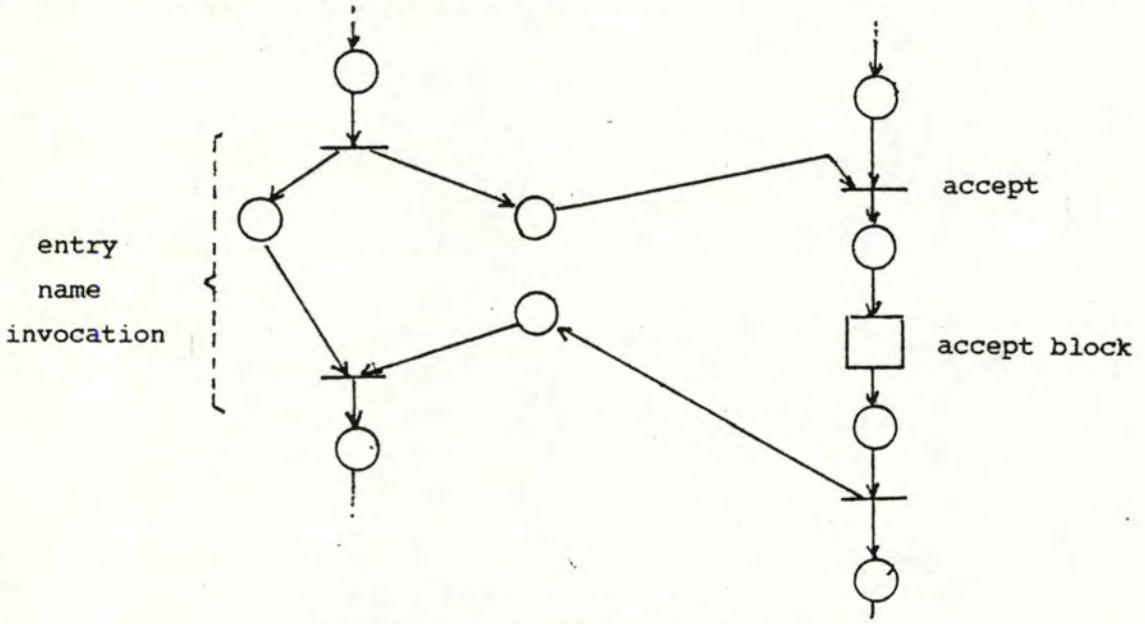


Figure 2.15: An ADA "Rendezvous"

The select can be illustrated with the bounded-buffer producer/consumer problem [2,21]. The bounded buffer and the operations allowing to insert and remove elements are implemented by an ADA task as follows:

```
task body boundedbuffer is
    buffer : array[0..9] of item;
    in,out : integer;
    count  : integer;
    in     := 0;
    out    := 0;
    count := 0;
begin
    loop
         select
             when count < 10 =>
                 accept insert (it : in item)
                     do buffer[in mod 10] := it end;
                 in := in + 1;
                 count := count + 1;
             or when count > 0 =>
                 accept remove (it : out item)
                     do it := buffer[out mod 10] end;
                 out := out - 1;
                 count := count - 1;
         end select;
    end;
end.
```

This task can be modeled by the Petri net in Figure 2.16. In this Petri net, place p1 represents the number of empty slots in the buffer (for the "when count < 10") and place p2 the number of used slots. Transition t1 corresponds to "when count < 10 => accept insert ...". Place p4 indicates that the task is executing the accept block (do..end). Place p3 assures the mutual exclusion between more accepts. Places p6 and p8 are marked if another task has invoked one of these entries and p7 and p9 represent the acknowledgment send to the calling task.

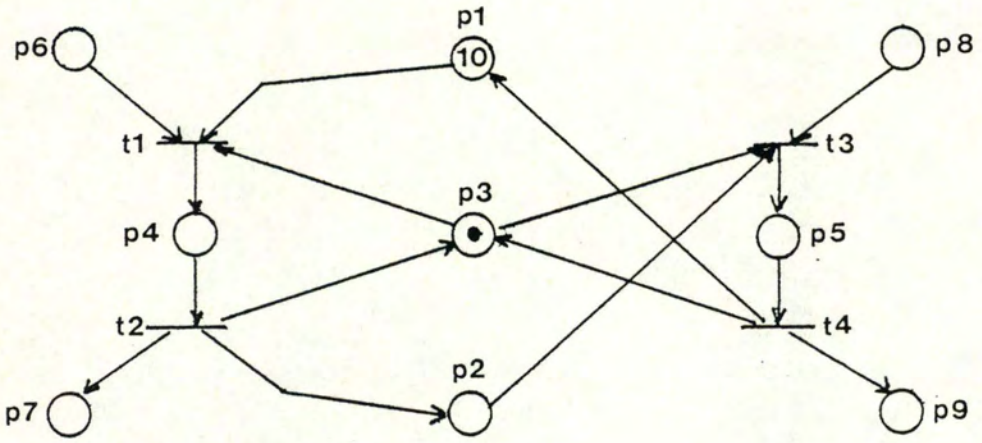


Figure 2.16: A bounded-buffer task

CHAPTER 3: ANALYSIS OF PETRI NETS

In this chapter, a method for analyzing Petri nets is described. First of all, the different analysis problems are stated. Then a technique for analyzing a Petri net is given and it will be shown how the different analysis problems can be solved using this method and which of the problems stated can be solved (as this method doesn't provide a solution to all problems). In a last section, some other analysis techniques are mentioned.

1. Analysis Problems

1.1. Safeness

A place in a Petri net is said to be safe if the number of tokens in that place never exceeds one. A Petri net is safe if all its places are safe. A Petri net in which places represent conditions must be safe because a condition can be true (place contains 1 token) or false (place contains 0 tokens). Multiple tokens on a place could lead to misbehavior in the Petri net.

The property of safeness is also very important in the modeling of hardware devices constructed with binary elements. Each binary element can represent the value zero or one.

A definition of the safeness property can be formulated as follows: A place $p_i \in P$ of a Petri net $PN = (P,T,I,O)$ with initial marking is safe if for all $M' \in [M]$, $M'(p_i) \leq 1$. A Petri net is safe if each place in the net is save.

Figure 3.1 is a Petri net which is not safe. Places p1 and p2 are safe but not place p3.

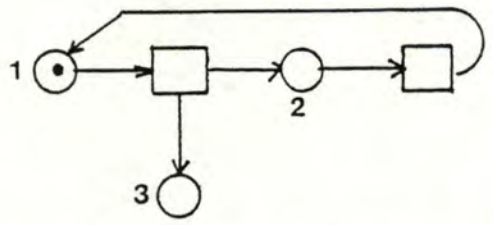


Figure 3.1: An unsafe Petri net

1.2. Boundedness

If the Petri net is constructed such that there can be at most k tokens on a given place, then this place is said to be "k-bounded". The bound k on the number of tokens can be a function of the place, i.e. different places can have different bounds. If a place is k-bounded, then it is also bounded for each $k' \geq k$.
Using this property it can be observed that a Petri net is k-bounded if all places are k-bounded. The bound for the Petri net is the maximum value of the bounds of each place. If the exact value of k is unknown, but is known to be some finite number, then the net is just referred to as being "bounded". Safeness is a special case of boundedness with k=1. The net in Figure 3.1 is not bounded since place p3 can hold an infinite number of tokens.

Boundedness is a very important property especially on single places. In the modeling of the bounded buffer problem we must verify that the number of tokens on the place representing the number of elements in the buffer never exceeds the bound for that place i.e. the capacity of the buffer. For the readers-writers problem, there are bounds imposed on several places: the number of processes reading must not exceed a certain number, the number of processes writing must not exceed one i.e. the place must be safe.

1.3. Conservation and Invariants

Another property that might be important is conservation of tokens. If tokens are used to represent resources, we would like to show that these tokens are neither distroyed nor created since the resources they represent are neither destroyed nor created. A Petri net is strictly conservative if the number of tokens in the net remains the same :

given a Petri net PN = (P,T,I,O) with the initial marking M_0, the Petri net is strictly conservative for all $M' \in [M0]$ if and only if

$$\sum_{p_i \in P} M'(p_i) = \sum_{p_i \in P} M_0(p_i)$$

The strict conservation is too restrictive since the number of input places and the number of output places of one transition must be the same.

We notice that not all tokens in a Petri net represent resources. Some tokens represent resources, others represent a particular value of the instruction pointer of a process, etc. So it would be interesting to be able to distinguish between different tokens. But we can only identify a token by its position on a place. That is why a weighting vector can be associated to the Petri net. The weighting vector gives a weight for each place and that weight is multiplied with the number of tokens on that place before summing up the number of tokens.

A Petri net is conservative with respect to a weighting vector $w = (w_1,..,w_n)$ $n=|P|$ if for all $M' \in [M_0]$

$$\sum_i w_i * M'(p_i) = \sum_i w_i * M_0(p_i)$$

A strictly conservative Petri net is conservative with respect to a weighting vector (1,1,..,1). All Petri nets are conservative with respect to (0,0,..,0).

Because all Petri nets are at least conservative with respect to one weighting vector, it is said that a Petri net is conservative if it is

conservative with respect to a non-zero weighting vector, $w > 0$ ($w_i > 0$).

A Petri net conservative with respect to a weighting vector is a Petri net satisfying an invariant, the weighting vector being called the invariant.

Another kind of invariant can be applied to the product of the number of tokens on two places. For a product invariant it will be verified that the product of the token numbers of two different places is zero. And this must be the case for each combination of two places corresponding to non-zero components of the product invariant. With a product invariant, it can be verified that two places are never marked at the same time.

1.4. Liveness

A transition t is potentially firable in a marking M if there exists a marking M' in the marking class of M under which t is enabled:

$$\exists\ M' \in [M] : M'[t\rangle$$

A transition is called live at a marking M if it is potentially firable in every marking in the marking class of M:

$$\forall\ M' \in [M] \quad \exists\ M'' \in [M'] : M''[t\rangle$$

Transition t is called dead at (under) M' if t cannot be activated under any marking of the marking class of M':

$$\forall\ M'' \in [M'] : \neg\ (M''[t\rangle)$$

The marking M' is then called t-dead. The transition t is thus not live if and only if there exists a marking $M' \in [M]$ such that t is dead at M' .

A marking M is called dead if all transitions are dead under M.

The Petri net PN is called live if and only if each transition is live at the intitial marking M_0.

Thus, if a Petri net has a dead marking, the system represented by the net can run into a state where the whole system cannot proceed: there occurs a deadlock-situation. Figure 3.2 illustrates this

problem. After the firing of transition t3, the net is in a dead marking and no transition can fire. If the Petri net is not live, it can run into a state where a transition can never be fired again, i.e. part of the system cannot proceed.

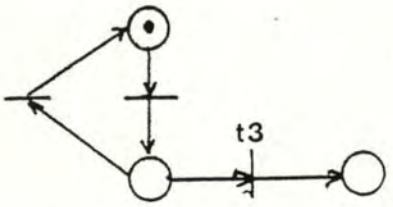


Figure 3.2: A net which can run into a deadlock

An example of a deadlock situation is a situation in which two processes P1 and P2 need two resources A and B. Each process obtained one resource. Now the two processes are each waiting for the other to release the second resource it needs to continue. Thus, the two processes are blocked, each waiting for the other.

1.5. The reachability problem

The reachability problem can be stated as follows: "given a Petri net and an initial marking M_0, is $M \in [M_0]$?"

Thus a marking M_k is called reachable from a marking M_0 if and only if there exist transitions $t_1,..,t_k$ and markings $M_1,..,M_k$, such that the firing of transition t_i produces the marking M_i out of the marking M_{i-1} (i:1..k).

This problem is particulary important because many analysis questions can be expressed in terms of reachability. For instance, Hack [9] has shown that the liveness problem is reducible to the reachability problem and that in fact the two problems are equivalent, since reachablility is also reducible to liveness.

Another problem is the coverability problem: given a Petri net with an initial marking M0 and a marking M', M' is coverable if and

only if

$$\exists \; M'' \in [M_0] : M'' \geq M'$$

2. Analysis technique (Reachability tree)

It is useful to represent the elementary changes of markings by a reachability tree. The reachability tree represents the reachability set of a Petri net. The nodes of the tree are reachable markings $M \in [M_0]$, and the arcs are labeled by the transitions which cause the marking changes.

This tree can be constructed as follows :

1. let the initial marking M_0 be the root of the tree;
   let the root be the current node;
2. for each transition t enabled at the current marking M:
   - create a new node with the marking M' such that M[t>M',
   - create an arc from the current node to the new node and label the arc with t;
3. repeat the second step for all newly created nodes.

It is obvious that this tree can be infinite if the net has unbounded capacity (an infinite number of tokens can be accumulated on a place) or if a marking is reproducible (M[->M). This is illustrated in Figure 3.3 which shows the reachability tree of the unbounded buffer model (Figure 2.6).

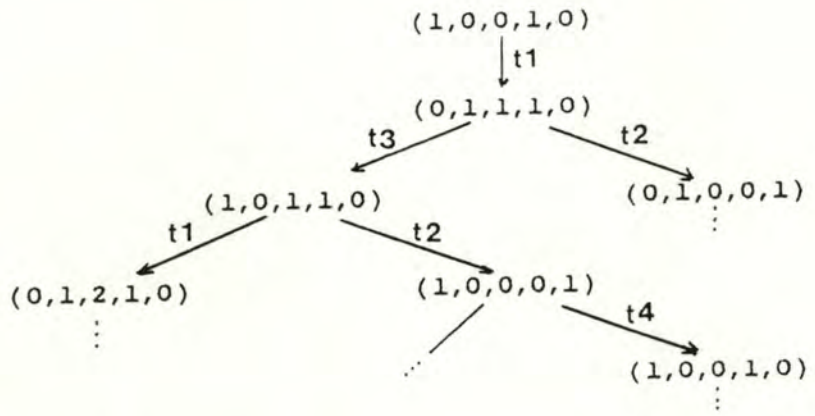


Figure 3.3: The reachability tree of the unbounded buffer

It can be observed that the sequence of transitions t1,t3 can be fired as ofter as wanted increasing the number of tokens on place p3. Consequently, an infinite number of tokens can be accumulated on place p3. The initial marking is reproducible by firing for instance the sequence t1,t3,t2,t4, causing the generation of an infinite number of nodes in the reachability tree.

If we want to use the reachability tree for analysis of Petri nets, we must modify our procedure in order to obtain a finite tree. This reduction of the tree is helped by dead markings because their marking class consists of the singleton {M} if M is the dead marking. Thus, for dead markings no further nodes will be generated in the reachability tree and the node will be called a terminal node and constitutes a leaf of the tree.

Another class of leaves consists of the nodes having a marking that appears already in the reachability tree and for which the marking class has already been generated. It is not necessary to generate the marking class once again for the new node because it will be the same as for the one already encountered. This node will be said to be a duplicate node in the reachability tree and it will not be considered anymore in the reachability tree construction.
One final means to cut down the reachability tree to a finite

representation is based on the observation that often two markings $M \in [M_0]$ and $M' \in [M]$, with $M < M'$ define a lot of different markings $\{M_1, M_2, \ldots\} \subseteq [M]$.
In this set M_{i+1} is obtained from Mi by the firing of the same transition sequence leading from M to M'.
Then, we have $M' - M = M_{i+1} - M_i > 0$.
This firing sequence can be repeated over and over, increasing the number of tokens in some place of the net.
In the reachability tree construction procedure, this subset of markings will be reduced to one node in the reachability tree and the special symbol ω (omega) is used to designate an infinite number of tokens.

The definition of ϵ can be given by the following properties : for all $z \in Z$:

$$\omega + z = \omega - z = \omega$$

$$z < \omega$$

$$\omega \leqslant \omega$$

$$0 * \omega = 0$$

Let $Z_\omega = Z \cup \{ \omega \}$.
For any two vectors x,y of Z_ω, the relations and operations x+y, x-y, x=y, $x \leqslant y$ are understood componentwise. The relation $x < y$ however is satisfied if and only if $x \leqslant y$ and $x \neq y$.

The precise algorithm for the reachability tree construction can now be stated. Each node is of one of the following types: terminal, duplicate, interior, frontier. Interior nodes are nodes already processed by the algorithm and which are neither terminal nor duplicate. A frontier node is a node not yet processed. To each node is associated a marking with $M(p_i) \in N_\omega$.

The algorithm is the following :

```
let the tree consist of one node, the root r;
declare r frontier;
while there are frontier nodes do
    choose a frontier node to process x;
    if there exists another node y in the tree
       which is not a frontier node, and has
       the same marking M[x]=M[y],
      then if y is of type terminal
             then declare x terminal
             else declare x duplicate
            fi
      else if no transitions are enabled at M[x]
             then declare x terminal
             else for each transition t : M[x][t› do
                    create a new node z;
                    M[z] := fire (t,M[x] );
                    if there exists a node y
                       on the path from the root
                       to x with M[y] ‹ M[x]
                     then for each M[y]i ‹ M[z]i do
                            M[y]i := ω;
                          od
                    fi
                    direct an arc labeled t from x to z;
                    declare z frontier;
                  od
                  declare x interior;
            fi
    fi
od.
```

The reachability tree of Figure 3.4 is shown in Figure 3.5

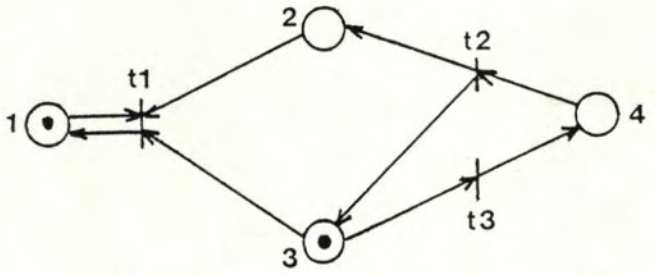


Figure 3.4: A Petri net with marking (1,0,1,0)
and infinite state-space

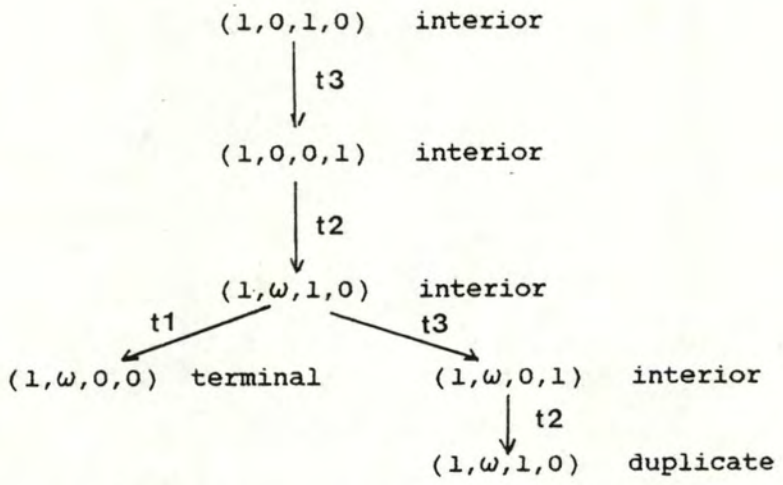


Figure 3.5: The reachability tree of the Petri net
in Figure 3.4

For the reachability tree constructing algorithm to be useful it is very important that it terminates. To prove this, it must be shown that the reachability tree is finite. Then the algorithm cannot continue to create frontier nodes forever. Since this dissertation stresses on the application of results obtained in pure Petri net theory, the prove is not given here. A proof can be found in Peterson [20].

3. Resolution power of the reachability tree

In this section, the resolution power of the reachability tree is discussed. For the decidable problems, we indicate how to solve them. Then, the limitations of the reachability tree are discussed.

3.1. Safeness and Boundedness

The safeness and the boundedness problems are decidable using the reachability tree.

A Petri net is bounded if and only if the symbol ω never appears in the reachability tree, i.e. no place of the net can contain an unlimited number of tokens. If the symbol ω occurs in the reachability tree, there exists a firing sequence which can be repreated arbitrarily often to increase the number of tokens to infinity. The symbol ω indicates by its position the unbounded place(s). Thus, a place in a Petri net is bounded if there is no marking in the reachability tree such that the component corresponding to the place is ω. The boundedness problem and the submarking boundedness problem can be decided by inspection of the reachablility tree.

The safeness problem can also be decided by inspection of the reachability tree. If there is no marking in the tree with the component corresponding to a given place greater than one, then the place is safe.

An interesting property is that of submarking boundedness. Even if a net is not bounded, some places can be bounded and that may sometimes suffice to verify the correct functioning of the net.

In the bounded buffer problem, for instance, if the place representing the number of elements in the buffer is bounded to the capacity of the buffer, the buffer will never overflow.

In the readers/writers model, the place representing the number of

processes writing must be safe (at most one writes).

These properties can always be verified on the reachability tree, even when the whole net is unbounded.

If the Petri net is bounded, it represents a finite state system and the reachability tree contains all reachable markings. The reachability tree represents the whole state space of the system and all other analysis questions can be solved by the inspection of the tree.

3.2. Conservation and Invariants

Conservation can be tested using the reachability tree. The weighted sum can be computed for each marking and all the sums can be compared for equality. If the sum is the same for all the markings, the Petri net is conservative with respect to the given weigths vector. If the sums are not equal, the net is not conservative.

If the net is not bounded, the weigths associated to the unbounded places must be zero, else the net is not conservative.

Thus, we can verify if a net is conservative with respect to a weighting vector. But we can also use the reachabliy tree another way round. The reachbility tree can be used to determine if a Petri net is conservative by finding a weighting vector. As defined earlier, a Petri net is conservative if it is conservative with respect to a strictly positive weigths vector. This imposes the boundedness of the net. If the net is conservative, a weighted sum S and a weights vector $w=(w_1, w_2, .. w_n)$ exist. For each reachable marking M, we have :

$$w_1 * M(p_1) + w_2 * M(p_2) + \ldots + w_n * M(p_n) = S$$

This defines a set of k linear equations in n+1 unknowns if the reachability tree contains k nodes. If we add to this the constraints

$$w_i > 0, \quad i:1,2,..,n$$

we have a well defined problem which can be solved.

Example : let us consider the mutual exclusion problem. Figure 3.6

gives the model and the corresponding reachability tree.

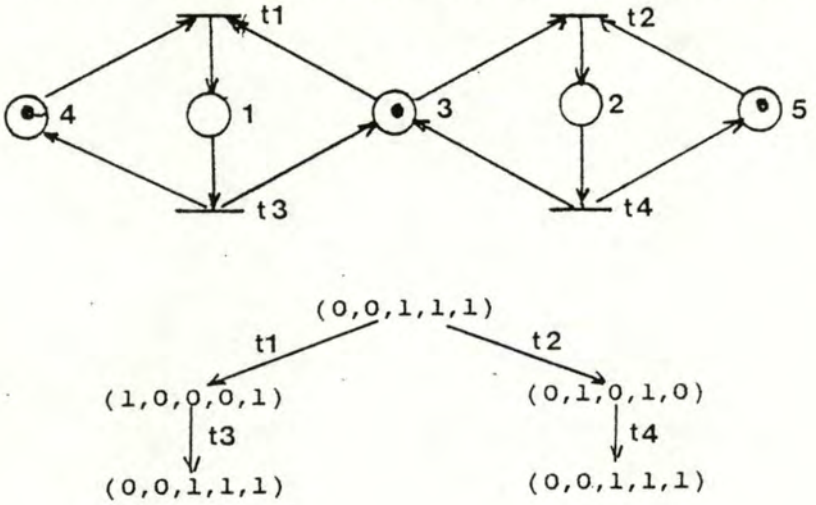


Figure 3.6: The mutual exclusion problem and the corresponding reachability tree

The system of equations is thus the following :

$w3 + w4 + w5 = S$

$w1 + w5 = S$

$w2 + w4 = S$

$wi > 0,\ i = 1..5$

A solution to this system is :

$w1 = 2,\ w2 = 2,\ w3 = w4 = w5 = 1$

The invariant on the sum of markings is the same as the conservation with respect to a weigthing vector with possibly some negative or zero weights. In the bounded buffer problem (Figure 2.7), for example, the sum of the two places representing the number of empty buffers and the number of full buffers must always be equal to the number of buffers n.

The product invariant is used to verify the mutual exclusion. It suffices to show that mutually exclusive places (e.g. critical sections) are never marked at the same time, i.e. the product of the markings of this places is zero for each marking in the reachability

tree. This shows then that if one process is in its critical section (place marked), the other one is not (place not marked).

3.3. Coverability

Given a marking M, is there a reachable marking M' which covers M, i.e. such that M' is greater or equal than M? This problem can be solved by inspection of the reachability tree.

3.4. Limitations of the reachability tree

The two problems of liveness and reachability can not in general be solved with the reachability tree. If the Petri net is unbounded, the tree contains omegas and there is some loss of information.

But although the reachability tree does not solve this problems in general, sometimes it does. If there is a terminal node in the reachability tree, it can be concluded that the net is not live. For the reachability problem, it may be the case that the marking is in the tree and then it is obviously reachable. If a marking is not covered, then it is not reachable.

If the tree contains no omegas, all reachability and liveness problems can be solved using the reachability tree. Sometimes, it is possible to modify a model in order to make it bounded; e.g. the unbounded buffer problem is transformed into a bounded buffer problem.

Another problem with the reachability tree construction is that even if the Petri net is bounded, the reachability tree can become very big. In such cases, the space or the computational power required to build the tree are sometimes too large for the analysis to be usefull.

4. Other analysis techniques

This section will introduce some other analysis techniques used to verify properties of Petri nets. These techniques are not developed within the framework of this dissertation. They are only mentioned to signal that there exist other analysis techniques for Petri nets.

4.1. Linear algebra

The Petri net analysis with linear algebra studies essentially structural properties of Petri nets, i.e. properties not depending on the initial marking of the net. This is done by looking into the structure of the incidence matrix C. The incidence matrix is obtained by subtracting the input matrix I from the output matrix O : C = O - I.

We observe that the passage from a marking M to a marking M' by the firing of a transition sequence s can be represented by the following equation:

$$M' = M + C \cdot s' .$$

s' is a column vector and each of its components stands for the number of times the corresponding transition fires in the firing sequence.

4.2. Reductions of nets

The Petri net models are sometimes too complex to be analyzed by the available techniques. This is often the case for the reachability construction. In particular cases, the complexity can be reduced by eliminating some aspects not relevant for the property to verify. Reductions of a Petri net, provided that they preserve its properties, may then be used to obtain a new net which suits to the analysis by reachability tree construction. For the linear algebra method, reductions can reduce the size of the matrixes to manipulate. A

reduction can consist of place substitution or suppression of transitions.

4.3. Petri net classes

There exist extensions to Petri nets and subclasses of Petri nets. The class of extended Petri nets is characterized by a greater modeling power than general Petri nets. Because of the extensions, some analysis methods for Petri nets cannot further be applied to this class of nets. This leads to a lower decision power for extended Petri nets. An example for this class are nets with inhibitor arcs. An inhibitor arc is an arc that enables a transition only if the incoming places are not marked.

Subclasses of Petri nets often come to live due to the observation that the modeling of some classes of systems don't require the whole modeling power of Petri nets. This leads to classes of Petri nets with a restricted modeling power, but with an increased decision power. Examples of subclasses of Petri nets are state machines, marked graphs and free-choice Petri nets.
State machines are Petri nets such that each transition has exactly one input and one output place and the arcs connecting it to these places are of value 1. A marked graph is a Petri net in which each place is an input for exactly one transition and an output for exactly one transitions. A free-choice Petri net is a Petri net in which each arc is labeled by 1 and if two or more transitions have an input place in common, they share all their input places.

CHAPTER 4: A PETRI NET ANALYSIS PROGRAM

This chapter describes a tool for anlyzing Petri nets. This tool is not a complete Petri net analysis package, but implements only one analysis method, the one described in this dissertation. It seemed important to me to enforce the present analysis method with a software tool showing that the theory presented is directly applicable. In a first section, an overall description of the software tool will be given. The following sections give a functional analysis of the program and some implementation details.

Notice that the program is written in the VAX-11 PASCAL programming language which runs under VMS. The program does not make use of special features of the VAX-11 PASCAL implementation in order to enforce understandability and portability.

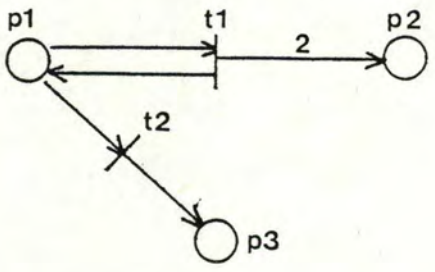
1. Overall description of the tool

The program is based on the reachability tree analysis technique. Once a Petri net has been input and the initial marking has been set, the reachability tree of the net can be constructed. After the construction of the reachability tree, different properties can be verified on the reachability tree.

The input of the Petri net consists in the decoding of a Petri net model stored previously in a text file. The model is a set of formulas describing the transitions. A transition is defined by the effect it has on the token load of places. If transition t1, when fired, removes one token from place p1 and puts 2 tokens on place p2, this will be expressed by the following formula:

t1=-p1+2p2

To illustrate the use of this modelisation language, Figure 4.1 shows a Petri net and the corresponding model.



t1=-p1+p1+2p2
t2=-p1+p3

Figure 4.1: A Petri net and its model

After having decoded a model, the user can set the initial marking for the Petri net. Once the marking set, the reachability tree can be constructed. For the same model, the process of setting an initial marking and building the reachability tree can be repeated an arbitrary number of times. This permits to analyze different scenarios depending on the initial marking of the Petri net.

The reachability tree can be displayed at the terminal or be printed to an output file. This possibility of having the output of the system displayed at the user terminal or printed to file exists for all analysis tasks.

The analysis questions a user can ask are the following :

1. boundedness of the Petri net,
2. deadlock,
3. coverability of a given marking,
4. reachability,
5. invariance on the sum of markings (conservation),
6. invariance on the product of markings (mutual exclusion).

This functions will be described in more detail in the next section.

2. Functional analysis

This section gives an overview of all the functions performed by the Petri net analysis tool and describes the effect of each function.

2.1. Input of a new model

This function allows a user to input a Petri net from a model file. The system asks for the name of the model and then decodes the model contained in that file. The name of the model file must be of type (extension) ".MOD". When asked for the file name, the user must not supply the file type (".MOD"), the system adds it automatically to the file name if it is not given. The model must respect the syntax given in Appendix A.
If there are errors in the model, the decoding is aborted and an appropriate message is given. When the decoding succeeds without error, the number of transitions and places in the Petri net are communicated to the user.

2.2. Construction of the reachability tree

This task constructs the reachability tree of the Petri net given an initial marking. If no model is present, the task is aborted after having printed an appropriate message to the user. If there is no initial marking, the user is asked to introduce one.

After having verified the initial conditions, the reachability tree is constructed according to the algorithm given in chapter 3. The user is kept informed on the progress of the reachability tree construction by printing one dot (`.`) for each creation of a new node. Once the reachability tree construction finished, the system displays the number of nodes in the tree and terminates the task.

2.3. Modification of the initial marking

This function allows the modification of the initial marking. The user can introduce another initial marking in order to construct the reachability tree with this marking. This feature makes it possible to analyze different scenarios depending on the initial marking of the Petri net.

2.4. Direction of output

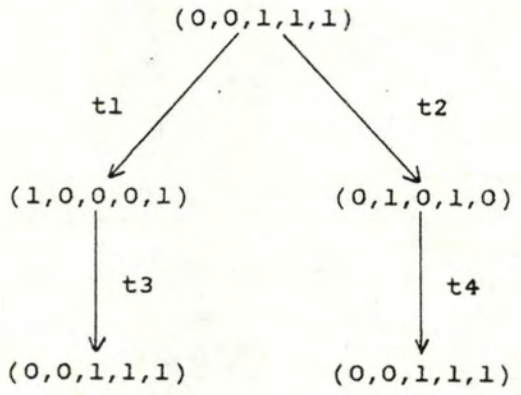
The output of the reachability tree and the analysis results can be directed to the user terminal or to a file that can be printed after the session. If the output is directed to a file, the name of the file will be given to the user. This name is "PNOUT.RES" if no model file has been read in. If a Petri net has already been introduced from a model file, the filename of the model file will be taken and the extension (file type) will be ".RES". Thus, if the model file "MUTEX.MOD" has been input and afterwards the output is directed to file, the file containing the results will have the name "MUTEX.RES".

2.5. Printing the reachability tree

If a reachability tree has already been constructed, it is output to the user terminal or to the output file if the output is directed to a file.

First, the number of nodes of the reachability tree is printed. The reachability tree is then printed level by level. In other words, for each marking, starting with the initial marking, all its follower markings are printed, prefixed by the type of the node (interior,

duplicate or terminal) and by the transition leading to it. Figure 4.2 shows the reachability tree of the mutual exclusion problem and the corresponding printout.




```
(0,0,1,1,1)
i t1 : (1,0,0,0,1)
i t2 : (0,1,0,1,0)

(1,0,0,0,1)
d t3 : (0,0,1,1,1)

(0,1,0,1,0)
d t4 : (0,0,1,1,1)
```

Figure 4.2: A reachability tree

2.6. Stop the session

Stops the session and leaves the program closing the output file, if it exists.

2.7. Query results

This is an entry point to a number of questions the user can ask about the properties of the Petri net. Thus, this function proposes a menu with different choices and leads to one of the functions described in the sequel.

After having answered a question, the system returns to this menu and the user can get out of it making the choice "E exit" to return to the first level.

2.8. Boundedness

This function gives an answer to the global boundedness of the Petri net. In either of the two cases, the bounds vector is given. This vector gives the bound for each place of the Petri net. Thus, the submarking boundedness problem can be answered for a given set of places using this bounds vector.

2.9. Deadlock

This function detects whether the system modeled by the Petri net can run into a deadlock state or not. A deadlock state is represented by a reachable marking at which no transition is enabled. If the system can run into a deadlock state, we are sure that the system is not live. If there are some dead markings, the system will signal it and the user can request a list of these markings.

2.10. Coverability

This function can be invoked to know if a given marking is covered by any other marking of the reachability tree. The user must introduce the marking for which to decide if it is coverable and the system will then decide if it is coverable. If this is the case, the first marking found in the reachability tree that covers the given one is printed.

2.11. Reachability

For a bounded Petri net, the reachability problem is decidable and this function decides whether a given marking is reachable or not.

In the case of an unbounded Petri net, the problem is more delicate. If the marking is in the reachability tree, obviously it is reachable. If it is not in the reachability tree and is not coverable by any other marking of the reachability tree, then we are sure that it is not reachable. In all other cases, the program cannot decide whether the marking is reachable or not.

2.12. Invariance on the sum

This function verifies the invariant on the sum of the markings. For each marking in the reachability tree the weighted sum of the marking is calculated and compared to the sum given by the user until a marking is found that does not satisfy the equality. The user gives the weights vector and the sum. If a marking is found that does not satisfy the condition (weighted sum = given sum), this marking is printed.

2.13. Invariance on the product

This function verifies a product invariant on all the markings. The user gives in a vector specifying the places to be considered in the verification. For all the considered places it is then verified that the marking product of all pairs of places is zero.

For example, if the places p1,p2 and p5 are considered, it will be verified that for each marking M in the reachability tree :

M(p1) * M(p2) = 0 and
M(p1) * M(p5) = 0 and
M(p2) * M(p5) = 0

If a marking is found in the reachability tree which does not satisfy this condition, it will be printed.

Remark : This technique is not necessary to verify the mutual exclusion between two places. The problem will be solved more efficiently by solving the submarking reachability problem. In fact, if a submarking is reachable with the two places containing one token, the mutual exclusion is not assured. In the case of a mutual exclusion of more than two places, the product invariant technique is shorter to formulate. For the example given before, we would have to solve three times the submarking reachability problem.

3. Implementation details

In this section, some implementation issues for the program described in this chapter are discussed.

3.1. Data structures

3.1.1. The Petri net

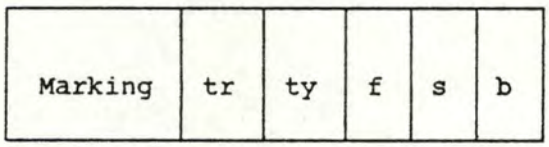
As already mentioned, the external representation of a Petri net is given by a model encoded in a modelisation language for which the syntax is given in Appendix A.
The internal presentation of the Petri net consists of two matrixes. These matrixes are the matrixes associated to the Input and the Output function respectively. Thus, the two matrixes, named inp and outp, are of size n * m, with n representing the maximum number of transitions and m the maximum number of places. Each transition in the Petri net is then defined by two rows in the two matrixes. Row t in the input matrix defines the input function for transition t and row t in the output matrix defines the output function of t.
To these two matrixes, we associate two integer variables representing the actual number of transitions and places of the Petri net.

Another representation would be the incidence matrix C of the Petri net. This matrix is obtained by subtracting the matrix inp form the matrix outp: C = outp - inp.
This matrix is often used in linear algebra analysis techniques for Petri nets, but doesn't fit to our use because such a representation dictates a restriction on the structure of the Petri net. This restriction is that a given place p cannot be an input and an output place of a transition. In fact, if transition t would have p as input and output place (with multiplicity 1), the subtraction of inp from outp would result in a value 0 for place p in the incidence vector for

transition t. This would mean that place p doesn't intervene in transition t, which is not the case.

3.1.2. The reachability tree

The reachability tree is represented as a collection of items connected with pointers. Each item contains the marking corresponding to the node, the number of the transition leading to that marking and the type of the node. The type can be interior, duplicate or terminal. Furthermore, an item contains pointers to the father, the son and the brother nodes. Thus, the structure of a node in the reachability tree is as shown in Figure 4.3. The reachability tree is given by a pointer to the root of the tree.



tr : transition leading to Marking
ty : type of node (i,d,f)
f : pointer to father
d : pointer to son
f : pointer to brother

Figure 4.3 : A node of the reachability tree

An example of a Petri net and the corresponding reachability tree is given in Figure 4.4.

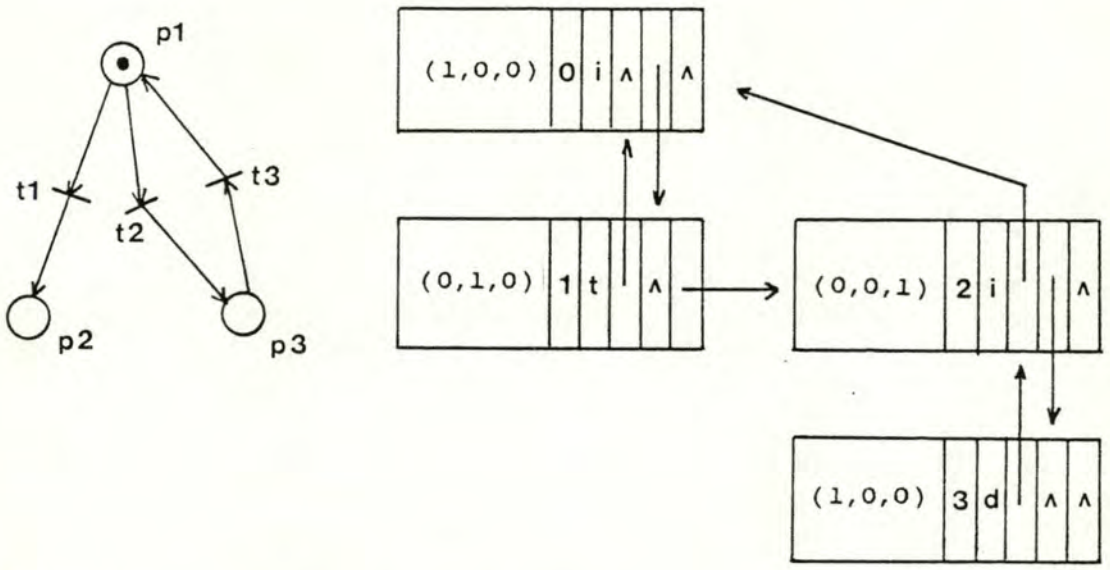


Figure 4.4 : A reachability tree

Notice that a father points to its first son only (if any), and not to all sons. This representation is choosen in order to economize in space. If we want all the pointers to the sons in the node, we would need an array of pointers to all sons of the node. This array would be of size n * size of pointer with n equals the maximum numer of transitions.

3.2. Algorithms

It is not my intention to describe all the algorithms used in the program but only some interesting one's.

3.2.1. The reachability tree construction

This is a refinement of the algorithm given in section 3.2, with some additional functions. The algorithm not only constructs the reachability tree, but computes the bounds for the different places. That is why the algorithm calls a function updating the bounds vector each time it encounters a new marking. The algorithm also builds up a list of all the terminal nodes.
This is done so because the boundedness and the deadlock questions are

the first questions asked about a Petri net. Thus, these two questions are answered once the reachability tree is build; i.e. the bounds vector contains the bounds and the list of terminal nodes contains all markings at which no transition is enabled. The algorithm also keeps track of the number of nodes in the reachability tree.

The algorithm is not reproduced here, but a Pascal implementation can be found in Appendix B.

3.2.2. Breath-first search

The algorithm searches through the whole tree until finding a node satisfying the criterion. The criterion is evaluated in another function. The search strategy is breath first. The algorithm is the one listed on the next page.

```
found := false;
if tree not empty
  then worklist := empty;
       resnode := root;
       repeat
          repeat
            if criterion (resnode)
              then found := true
              else if resnode has son
                     then add son at tail
                          of worklist
                   fi
                   resnode := brother of resnode
            fi
          until found or (resnode = nil);
          if not found
            then remove first from worklist and
                 associate it to resnode
       until found or
             (worklist was empty at last remove)
fi
```

CHAPTER 5: APPLICATIONS

In this chapter, I will show the usefulness of the tool described in the previous chapter by using it to analyze several systems of concurrent activities and by verifying their correct functioning with respect to their specification . First, the analysis tool is used to analyze some of the problems stated in chapter 2. Then, an extended send/receive model is presented and verified. All the models and the results from the analysis tool are listed in Appendix C.

1. The mutual exclusion problem

This problem is described in section 2.1.1. After the construction of the reachability tree, we know that the Petri net is bounded. Thus, all the analysis problems are solvable. First, it must be verified that at any moment at most one of the processes is engaged in its critical section. This can be done by verifying a Product Invariant on the places p1 and p2. After inspection of the reachability tree, it can be concluded that te invariant is verified. In other words, place p1 and place p2 are mutually exclusive. It follows that at most one of the two processes can be in its critical section.

Another property to verify is that stopping a process in the remainder of his cycle (not in its critical section) has no effect upon others. In fact, if we do not mark place p5 in the initial marking, the second process is not activated. But this does not prevent the first one from cycling and entering its critical section. This can be seen by observing the bounds for the different places. The bounds for place p1 and p4 are 1. Since the places represent dwell-points for the instruction pointer, it is verified that the process runs.

Notice that the Petri net cannot run into a deadlock state with this

initial marking.

An application with a process leaving its critical section without replacing a token on the synchronization place p3 would result in a deadlock. This model is given in Figure 5.1.

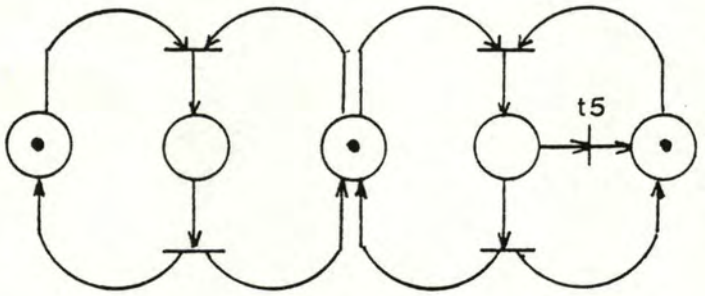


Figure 5.1: Incorrect mutual exclusion

Transition t5 would allow process 2 to leave the critical section without replacing a token on the synchronization place. In a program, this could be a branch instruction out of the critical section. The analysis of this Petri net shows that it can run into a deadlock state.

The general mututal exclusion problem can be verified using the Product invariant technique. For the Petri net presented in Figure 2.2, it can be shown that at most one of the four places standing for the critical section is marked at a time.

2. The Dining Philosophers

In section 2.1.2, three different Petri nets are given for this problem. The corresponding model files are given in Appendix C.

The analysis of the first of the three models shows that the Petri net can run into a deadlock state where each philosopher has picked up one fork and cannot continue, because there are no more forks on the table. In the dead marking, places p11 through p15 are marked and no other places. Places p11 through p15 represent the fact that the philosophers have picked up the first chopstick. Places p1 through p5 represent the chopsticks, p6 trough p10 stand for the philosophers meditating and p16 through p20 for the philosophers eating.

The second model cannot run into a deadlock state. This Petri net gives a correct solution to the Philosophers problem.

For the third solution, we have explained why the philosophers cannot starve. The starvation freeness cannot be verified with our tool, but we can verify that the solution is deadlock free.

3. The Sender/Receiver model [22]

An application is given where a sender and receiver are connected by a bounded capacity channel. The bound is set to 5 in this example. Each of the two processes can be in an active or in an inactive state. The receiver can only go into the inactive state if the sender is in the inactive state and if the channel is empty. To realize this synchronization, we introduce a second channel between the two processes. This channel is used to transmit the "finished" message of the sender to the receiver. The send and receive for the messages containing data are represented by transition t2 and t6 respectively. The send and receive for the "finished"-signal are

implemented by transition t4 and t8. Note that another condition to enable t8 is that the channel is empty. This condition is satisfied when place p5 holds n (5) tokens. Place p5 is the complement of place p4 representing the number of messages in the link. To execute this two processes, we have added some places and transitions to start up the two processes. This places represent the environment of the process. The model corresponding to this description is given in Figure 5.2.

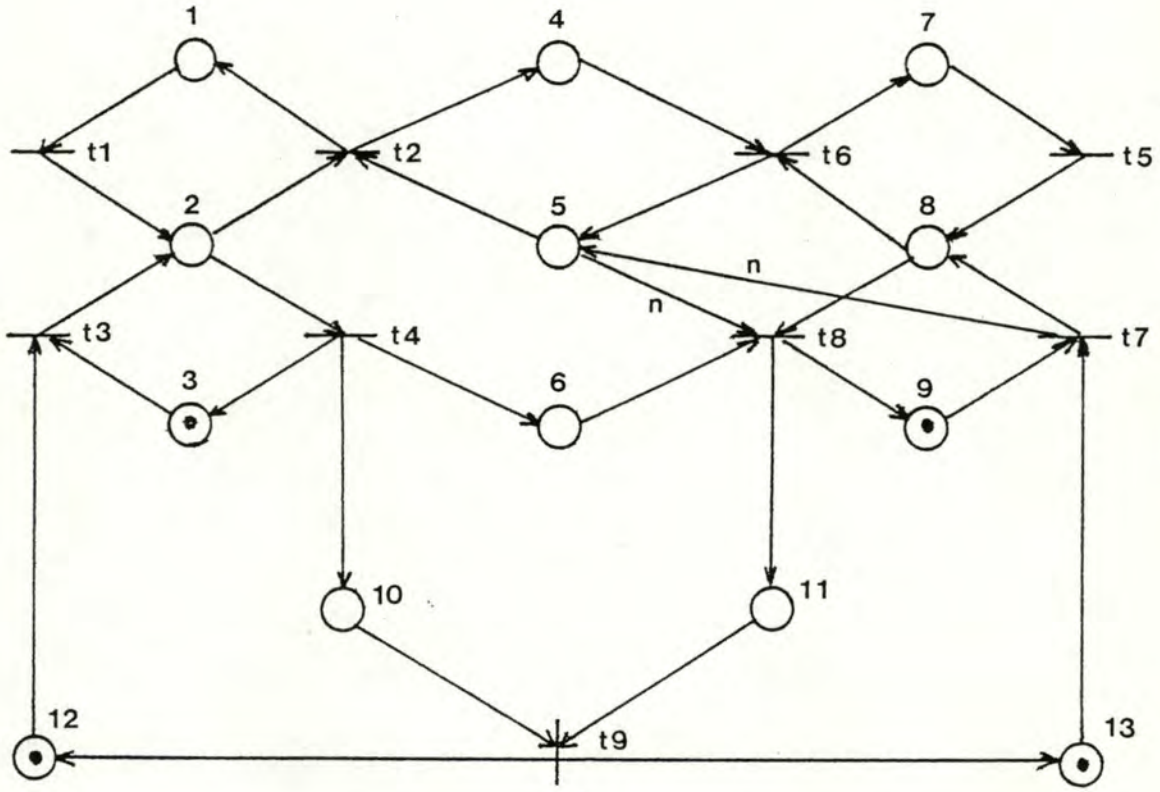


Figure 5.2 : Extended Sender/Receiver

If the send/receive system is correctly modeled, the model verifies the following properties:

1. Sender and receiver are always in one of the following states {p1,p2,p3} respectively {p7,p8,p9}. Place p1 and p2 stand for the sender being in an active state and in p3, the sender is

inactive.

2. The channel never contains more than 5 messages (tokens).
3. The sender (resp. receiver) is inactive if and only if he has given a corresponding signal to the environment. He can leave the inactive state only through a signal from the environment.
4. If the sender is in the inactive state, he can leave this state only when the receiver process is in his inactive state too.
5. The receiver's decision to receive or to go into an inactive state depends totally on the behavior of the sender.
6. The receiver can go into the inactive state only if the channel is empty and the sender is inactive.
7. The Petri net cannot run into a deadlock state.

All this properties will now be verified using the Petri net analysis tool. The construction of the reachability tree shows that it contains 76 nodes and that the Petri net is bounded. All places except the two representing the bounded channel (p4 and p5) are safe. The results can be found in Appendix C.

Property 1 can be verified using a sum invariant. The invariant for the sender is :

$$M(p1) + M(p2) + M(p3) = 1.$$

For the receiver, we have the invariant

$$M(p7) + M(p8) + M(p9) = 1.$$

This two invariants are verified on each of the markings of the reachability tree.

The second property is verified by looking to the bounds vector. The bound for place p4 is 5, i.e. the channel never contains more than five messages.

Property 3 is verified for the sender using the invariant

$$M(p10) + M(p12) - M(p3) = 0.$$

In other words, place p3 is marked only if place p10 or p12 is marked. The same property can be verified for the receiver with the invariant

$$M(p11) + M(p13) - M(p9) = 0.$$

Verifying the invariant

$$M(p6) - M(p10) + M(p11) = 0$$

shows that $M(p6) = 1$ implies $M(p10) = 1$ because a place cannot have a negative marking. Thus, if the sender is inactive, p10 is marked and this mark can only be removed by a firing of t9. But t9 is only enabled if the receiver is in the inactive state.

If property 5 is not satisfied, transition t6 and t8 can be enabled at the same time. This would require that $M(p4) \geq 1$, $M(p5) \geq 5$ and $M(p6) \geq 1$. It can however be shown on the reachability tree that such a marking is not covered, i.e. there can never be a conflict between the two transitions.

Property 6 is verified by the required token load for t8 to fire. The number of tokens on place p6 must be greater or equal than one and the marking of place p5 must be greater or equal than five.

The last property is also verified by the Petri net. The reachablity tree contains no dead marking.

CONCLUSIONS

In this dissertation, Petri nets were presented as a tool for modeling and analyzing systems of concurrent activities. The first chapter introduced common definitions of terms related to Petri nets and their execution. These concepts allowed us to model synchronization problems and mechanisms in chapter two. Chapter three gave an overview of important analysis questions and showed how to solve them using the reachability tree analysis technique. Since we wanted to analyze automatically the modeled systems, a Petri net analysis program was implemented. This software tool has been described in chapter four and some implementation details were discussed. Finally, chapter five showed how to put this method and the tool to work, verifying some modeled systems.

These observations suggest some conclusion. It has been shown that Petri nets are a good tool for dealing with the modeling and analysis of concurrent processes. The modeling of a lot of actually existing synchronization mechanisms is feasible and much of the analysis questions can be decided automatically.

Within the framework of this dissertation, only one analysis method has been presented in detail. This method has also been implemented to show the real usefulness of such a tool. A complete Petri net analysis package however would do much more. It would implement different analysis techniques, work on more classes of Petri nets and include a powerful graphics based net editor. Let me signal that there exist software packages implementing some of these features [3,4].

This conclusion should encourage us to deal in a structured and formal way with the construction of concurrent programs rather than reasoning informally or trying to debug the programs by testing.

BIBLIOGRAPHY

1. P. Azema, B. Berthomieu, P. Decitre :
The Design and Validation by Petri Nets of a Mechanism for the Invocation of Remote Servers.
Information Processing 80, North-Holland 1980, 599-604

2. M. Ben-Ari :
Principles of Concurrent Programming
Prentice-Hall 1982

3. G.W. Brams :
Reseaux de Petri : Theorie et Pratique
Masson 1983, 2 Vol's

4. E. Ciapessoni, M. Negri, D. Pieragostini :
Netlab : A Software Tool for Drawing and Validating Petri Nets
Special Interest Group "Petri Nets and related System models"
Gesellschaft fuer Informatik
Newsletter 18, Oct.84, 4-6

5. L.W. Cooprider :
Petri Nets and the Representation of Standard Synchronization
Carnegie-Mellon University, Jan.76

6. P.J. Courtois, F. Heymanns, D.L. Parnas :
Concurrent Control with "Readers" and "Writers"
Communications of the ACM, Vol.14, No.10, Oct.71, 667-668

7. E.W. Dijkstra :
Co-operating Sequential Processes
Programming languages, F. Genuys, ed.
Academic Press 1968, 43-112
(Reprint of Technical Report EWD-123, Technological University, Eindhoven, 1965)

8. E.W. Dijkstra :
Hierarchical Ordering of Sequential Processes
Acta Informatica, Vol.1, N.2, 1971, 115-138

9. M.H. Hack :
The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems
Proceedings of the 15th Annual IEEE Symposium on Switching Automata Theory, Oct.74

10. O. Herzog :
Zur Analyse der Kontrollstruktur paralleler Programme mit Hilfe von Petri-Netzen.
Universitaet Dortmund, Abteilung Informatik, Bericht Nr.24/76, 1976

11. C.A.R. Hoare :
Communicating sequential processes
Communications of the ACM, Vol.21, No.8, 666-677

12. R.C. Holt :
Concurrent Euclid, the Unix System, and Tunis.
Prentice-Hall 1982

13. M. Jantzen, R. Valk :
Formal Properties of Place/Transition Nets.
Lecture Notes in Computer Science 84, Springer 1980

14. R.M. Keller :
Vector Replacement Systems : A Formalism for Modeling Asynchronous Systems
TR 117, Computer Science Laboratory, Princeton University, 1972

15. R.M. Keller :
Generalized Petri Nets as Models for System Verification.
TR 200, Department of Electrical Engineering, Princeton University, Dec.75

16. K. Lautenbach, H.A. Schmid :
Use of Petri Nets for proving correctness of concurrent Process Systems.
Proceedings of the IFIP Congress 74, Amsterdam 1974, 187-191

17. H. Ledgard :
Ada - An Introduction
second edition, Springer 1983

18. D. May, R. Taylor :
Occam - an overview
Microprocessors and Microsystems, Vol.8, N.2, Mar.84, 73-89

19. J.L. Peterson :
Petri Nets
Computing Surveys, Vol.9, N.3, Sep.77, 223-252

20. J.L. Peterson :
Petri Net Theory and the Modeling of Systems
Prentice Hall 1981

21. J.L. Peterson, A. Silberschatz :
Operating System Concepts
Addison-Wesley 1983

22. W. Reisig :
Petrinetze - Eine Einfuehrung
Springer 1983

APPENDIX A: MODELING LANGUAGE

This appendix is a description of the modeling language used to represent a Petri net.

A Petri net model consists of declarations of transitions. Each transition is built up from a transition identifier (e.g. t1), separated from an expression by an equal sign.
An expression describes the effect of the transition on the token load of the places. It consists of the enumeration of all places intervening in the transition. If a place belongs to the input places of a transition, it is prefixed by the minus sign. The place identifier can also be prefixed by an integer value representing the label of the arc connecting the place and the transition, i.e. the number of tokens to remove or to put on the place.
A place identifier is composed of the letter "p" followed by the number of the place.
The complete syntax description follows. It is represented in Backus-Naur Form.

```
<petri-net>   ::= <transition>
                | <transition> <petri-net>

<transition> ::= <transid> <equal> {<sign>} <expr>

<expr>        ::= <factor>
                | <factor> <sign> <expr>

<factor>      ::= <placeid>
                | <unsigned> <placeid>

<transid>     ::= t <unsigned>

<placeid>     ::= p <unsigned>

<equal>       ::= =

<sign>        ::= + | -

<unsigned>    ::= <digit>
                | <digit> <unsigned>

<digit>       ::= 0..9
```

APPENDIX B: ALGORITHMS

This appendix contains the listing of two algorithms implemented in PASCAL. The first one is the reachability tree constructing algorithm. The second one implements the breath-first search. The test of the criterion for the search is embedded in a function passed as an argument to the search procedure. The listing of the entire program can be obtained from the author.

```
1153    procedure reach_tree;
1154
1155    (* function :   given a PN defined by I,O,noftrans,nofplaces and a          *)
1156    (*              root node, build the reachability tree                      *)
1157    (*                                                                          *)
1158    (* input    :   -                                                          *)
1159    (*                                                                          *)
1160    (* output   :   -                                                          *)
1161    (*                                                                          *)
1162    (* invokes  : breath_first                                                  *)
1163    (*            find_marking                                                  *)
1164    (*            update_bds_vector                                             *)
1165    (*            addhead                                                       *)
1166    (*            remove                                                        *)
1167    (*            enabled                                                       *)
1168    (*            initlist                                                      *)
1169    (*            print_dot                                                     *)
1170
1171    var     x,y,newnode,son : nodeptr;      (* work nodes                      *)
1172            i               : integer;      (* index for number of transitions *)
1173            fnd,                            (* boolean for result of search    *)
1174            term,                           (* indicator for terminal nodes    *)
1175            emptind         : boolean;      (* indicator of emptyness for the  *)
1176                                            (* list of frontier nodes          *)
1177            frontiers       : list;         (* list of frontier nodes          *)
1178
1179            wrkmark         : marking;      (* work variable for markings      *)
1180
1181    begin
1182            initlist (frontiers);
1183            nofnodes := 1;
1184            x := root;
1185            update_bds_vector(x^.mark);
1186            repeat
1187                 xmark := x^.mark;
1188                 breath_first (no_frontier,y,fnd);
1189                 if fnd
1190                   then if (y^.nodetype = terminal)
1191                          then x^.nodetype := terminal
1192                          else x^.nodetype := duplicate
1193                   else begin
1194                          term := true;
1195                          for i := 1 to noftrans do
1196                              if enabled (x^.mark,i)
1197                                then begin
1198                                       find_marking(x^.mark,i,x,wrkmark);
1199                                       update_bds_vector(wrkmark);
1200                                       nofnodes := nofnodes + 1;
1201                                       print_dot (nofnodes);
1202                                       new(newnode);
1203                                       with newnode^ do
1204                                       begin
1205                                           mark := wrkmark;
1206                                           trans := i;
1207                                           father := x;
```

```
1208                                                brother := nil;
1209                                                son := nil;
1210                                                nodetype := frontier;
1211                                            end;
1212                                            if term    (* first son of x *)
1213                                              then begin
1214                                                     term := false;
1215                                                     x^.son := newnode;
1216                                                     x^.nodetype := interior;
1217                                                end
1218                                              else son^.brother := newnode;
1219                                            son := newnode;
1220                                            addtail (frontiers,newnode);
1221                                        end;
1222                                if term (* no transition enabled *)
1223                                  then begin
1224                                         x^.nodetype := terminal;
1225                                         addtail (terminals,x);
1226                                       end;
1227                            end;
1228                    remove (frontiers,x,emptind);
1229            until emptind;
1230    end;
1231    (* reach_tree *)
```

```
1007        procedure breath_first (function criterion (tocheck : nodeptr) : boolean;
1008                                var resnode : nodeptr;
1009                                var found : boolean);
1010
1011        (* function :    breath-first search on the reachability tree.                    *)
1012        (*               returns found=true if node found for which criterion is          *)
1013        (*               satisfied (resnode points to this node).                         *)
1014        (*               returns found=false if no node in the reachability tree          *)
1015        (*               satisfies the criterion                                          *)
1016        (*                                                                                *)
1017        (* input :       criterion boolean functions which evaluates a criterion         *)
1018        (*               for the node given as argument.                                  *)
1019        (*                                                                                *)
1020        (* output :      resnode if found=true pointer to the element of the              *)
1021        (*                       reachability tree for which criterion is satisfied       *)
1022        (*                                                                                *)
1023        (*               found   true if node found which satisfies the criterion         *)
1024        (*                       false otherwise                                          *)
1025        (*                                                                                *)
1026        (* invokes :     criterion                                                        *)
1027        (*               addtail                                                          *)
1028        (*               remove                                                           *)
1029
1030
1031        var     wrklst  : list;     (* list of nodes to be considered later on       *)
1032                isempty : boolean;  (* used as emptyness indicator for wrklst        *)
1033
1034        begin
1035            found := false;
1036            if root <> nil
1037              then begin
1038                       initlist (wrklst);
1039                       resnode := root;
1040                       repeat
1041                            repeat
1042                                if criterion (resnode)
1043                                  then found := true
1044                                  else begin
1045                                         if (resnode^.son <> nil)
1046                                           then addtail (wrklst,resnode^.son);
1047                                         resnode := resnode^.brother;
1048                                       end;
1049                            until found or (resnode = nil);
1050                            if (not found)
1051                              then remove (wrklst,resnode,isempty);
1052                       until found or isempty;
1053                   end;
1054        end;
1055        (* breath_first *)
```

APPENDIX C: ANALYSIS RESULTS

This appendix contains the analysis results of the cases studied in Chapter 5.

1. The mutual exclusion problem

The model file (MUTEX.MOD) :

```
t1=-p4-p3+p1
t2=-p5-p3+p2
t3=-p1+p3+p4
t4=-p2+p3+p5
```

The analysis results for this Petri net are listed below:

REACHABILITY TREE

```
Number of nodes :          5

(0,0,1,1,1)
1 t1    : (1,0,0,0,1)
1 t2    : (0,1,0,1,0)

(1,0,0,0,1)
d t3    : (0,0,1,1,1)

(0,1,0,1,0)
d t4    : (0,0,1,1,1)

All places of the Petri net are bounded.
Here are the bounds :
(1,1,1,1,1)

The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node

 The product invariant :
(1,1,0,0,0)
is verified for all markings
```

REACHABILITY TREE

```
Number of nodes :          3

(0,0,1,1,0)
i  t1    : (1,0,0,0,0)

(1,0,0,0,0)
d  t3    : (0,0,1,1,0)

All places of the Petri net are bounded.

Here are the bounds :
(1,0,1,1,0)

The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node
```

The model file of the incorrect mutual exclusion solution (MUTEXERR.MOD) and the analsis results :

```
t1=-p4-p3+p1
t2=-p5-p3+p2
t3=-p1+p3+p4
t4=-p2+p3+p5
t5=-p2+p5
```

REACHABILITY TREE

```
Number of nodes :          0

(0,0,1,1,1)
i  t1    : (1,0,0,0,1)
i  t2    : (0,1,0,1,0)

(1,0,0,0,1)
d  t3    : (0,0,1,1,1)

(0,1,0,1,0)
d  t4    : (0,0,1,1,1)
t  t5    : (0,0,0,1,1)

All places of the Petri net are bounded.

Here are the bounds :
(1,1,1,1,1)
The Petri net can run into a deadlock

Reason : there are terminal nodes in the reachability tree
         i.e. dead markings
```

LIST OF TERMINAL NODES

t5 : (0,0,0,1,1)

General mutual exclusion for 4 processes :

REACHABILITY TREE

Number of nodes : 9

(1,0,1,0,1,0,1,0,1)
i t1 : (0,1,0,0,1,0,1,0,1)
i t3 : (0,0,1,1,0,0,1,0,1)
i t5 : (0,0,1,0,1,1,0,0,1)
i t7 : (0,0,1,0,1,0,1,1,0)

(0,1,0,0,1,0,1,0,1)
d t2 : (1,0,1,0,1,0,1,0,1)

(0,0,1,1,0,0,1,0,1)
d t4 : (1,0,1,0,1,0,1,0,1)

(0,0,1,0,1,1,0,0,1)
d t6 : (1,0,1,0,1,0,1,0,1)

(0,0,1,0,1,0,1,1,0)
d t8 : (1,0,1,0,1,0,1,0,1)

All places of the Petri net are bounded.

Here are the bounds :
(1,1,1,1,1,1,1,1,1)
The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node

2. The Dining Philosophers

The incorrect solution looks as follows (PHILINC.MOD):

```
t1=-p1-p6+p11
t2=-p2-p7+p12
t3=-p3-p8+p13
t4=-p4-p9+p14
t5=-p5-p10+p15
t6=-p5-p11+p16
t7=-p1-p12+p17
t8=-p2-p13+p18
t9=-p3-p14+p19
t10=-p4-p15+p20
t11=-p16+p6+p5+p1
t12=-p17+p7+p1+p2
t13=-p18+p8+p2+p3
t14=-p19+p9+p3+p4
t15=-p20+p10+p4+p5
```

```
All places of the Petri net are bounded.

Here are the bounds :
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
The Petri net can run into a deadlock

Reason : there are terminal nodes in the reachability tree
         i.e. dead markings


         LIST OF TERMINAL NODES
         ----------------------


t5    : (0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0)
```

Correct solution for the 5 philosophers (PHILCOR.MOD) and analysis results:

```
t1=-p5-p1-p6+p11
t2=-p1-p2-p7+p12
t3=-p2-p3-p8+p13
t4=-p3-p4-p9+p14
t5=-p4-p5-p10+p15
t6=-p11+p6+p5+p1
t7=-p12+p7+p1+p2
t8=-p13+p8+p2+p3
t9=-p14+p9+p3+p4
t10=-p15+p10+p4+p5
```

REACHABILITY TREE

Number of nodes : 31

```
(1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)
i  t1    : (0,1,1,1,0,0,1,1,1,1,1,0,0,0,0)
i  t2    : (0,0,1,1,1,1,0,1,1,1,0,1,0,0,0)
i  t3    : (1,0,0,1,1,1,1,1,0,1,1,0,0,1,0,0)
i  t4    : (1,1,0,0,1,1,1,1,1,0,1,0,0,0,1,0)
i  t5    : (1,1,1,0,0,1,1,1,1,1,0,0,0,0,0,1)

(0,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0)
i  t3    : (0,0,0,1,0,0,1,0,1,1,1,1,0,1,0,0)
i  t4    : (0,1,0,0,0,0,1,1,0,1,1,1,0,0,1,0)
d  t6    : (1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)

(0,0,1,1,1,1,0,1,1,1,1,0,1,0,0,0)
i  t4    : (0,0,0,0,1,1,0,1,0,1,1,0,1,0,1,0)
i  t5    : (0,0,1,0,0,1,0,1,1,1,0,0,1,0,0,1)
d  t7    : (1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)

(1,0,0,1,1,1,1,0,1,1,1,0,0,1,0,0)
d  t1    : (0,0,0,1,0,0,1,0,1,1,1,1,0,1,1,0,0)
i  t5    : (1,0,0,0,0,0,1,1,0,1,1,0,0,0,1,0,1)
d  t8    : (1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)

(1,1,0,0,1,1,1,1,0,1,1,0,0,0,1,0,0)
d  t1    : (0,1,0,0,0,0,1,1,0,1,1,1,0,0,1,0,0)
d  t2    : (0,0,0,0,1,1,0,1,0,1,1,0,1,0,1,0,0)
d  t9    : (1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)

(1,1,1,0,0,1,1,1,1,1,0,0,0,0,0,1)
d  t2    : (0,0,1,0,0,1,1,0,1,1,1,0,0,1,0,0,1)
d  t3    : (1,0,0,0,0,1,1,1,0,1,1,0,0,0,1,0,1)
d  t10   : (1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0)

(0,0,0,1,0,0,1,1,0,1,1,1,0,1,0,0)
d  t6    : (1,0,0,1,1,1,1,1,0,1,1,1,0,0,1,0,0)
d  t8    : (0,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0)

(0,1,0,0,0,0,1,1,1,0,1,1,1,0,0,1,0)
d  t6    : (1,1,0,0,1,1,1,1,1,0,1,1,0,0,0,1,0)
d  t9    : (0,1,1,1,0,0,1,1,1,1,1,1,1,0,0,0,0)

(0,0,0,0,1,1,1,0,1,1,0,1,1,0,1,0,0)
d  t7    : (1,1,0,0,1,1,1,1,1,0,1,1,0,0,0,1,0)
d  t9    : (0,0,1,1,1,1,1,0,1,1,1,1,0,1,0,0,0)

(0,0,1,0,0,1,1,0,1,1,1,0,0,1,0,0,1)
d  t7    : (1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,1)
d  t10   : (0,0,1,1,1,1,1,0,1,1,1,1,0,1,0,0,0)

(1,0,0,0,0,1,1,1,0,1,1,0,0,0,1,0,1)
d  t8    : (1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,1)
d  t10   : (1,0,0,1,1,1,1,1,0,1,1,1,0,0,1,0,0)
```

```
All places of the Petri net are bounded.

Here are the bounds :
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node
```

A starvation free solution (PHILSTAR.MOD) that is shown to be deadlock-free:

```
t1=-p1-p21+p11
t2=-p2-p22+p12
t3=-p3-p23+p13
t4=-p4-p24+p14
t5=-p5-p25+p15
t6=-p5-p11+p16
t7=-p1-p12+p17
t8=-p2-p13+p18
t9=-p3-p14+p19
t10=-p4-p15+p20
t11=-p16+p6+p5+p1+p26
t12=-p17+p7+p1+p2+p26
t13=-p18+p8+p2+p3+p26
t14=-p19+p9+p3+p4+p26
t15=-p20+p10+p4+p5+p26
t16=-p26-p6+p21
t17=-p26-p7+p22
t18=-p26-p8+p23
t19=-p26-p9+p24
t20=-p26-p10+p25
```

```
The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node
```

3. The sender/receiver

The model file (SENDREC.MOD):

```
t1=-p1+p2
t2=-p2-p5+p1+p4
t3=-p3-p12+p2
t4=-p2+p3+p6+p10
t5=-p7+p8
t6=-p4-p8+p5+p7
t7=-p9-p13+p3+5p5
t8=-5p5-p8-p6+p9+p11
t9=-p10-p11+p12+p13
```

Verification of property 1 :

```
 The sum invariant :
(1,1,1,0,0,0,0,0,0,0,0,0,0)
with sum :                1
is verified for all markings
```

```
 The sum invariant :
(0,0,0,0,0,0,1,1,1,0,0,0,0)
with sum :                1
is verified for all markings
```

Verification of property 2 :

```
All places of the Petri net are bounded.

Here are the bounds :
(1,1,1,5,5,1,1,1,1,1,1,1,1)
```

Verification of property 3 :

The sum invariant :
(0,0,-1,0,0,0,0,0,0,1,0,1,0)
with sum : 0
is verified for all markings

The sum invariant :
(0,0,0,0,0,0,0,0,-1,0,1,0,1)
with sum : 0
is verified for all markings

Verification of property 4 :

The sum invariant :
(0,0,0,0,0,1,0,0,0,-1,1,0,0)
with sum : 0
is verified for all markings

Verification of property 5 :

The marking :
(0,0,0,1,5,1,0,0,0,0,0,0,0,0)
is not covered by any other marking

Verification of property 7 :

The Petri net cannot run into a deadlock
The tree doesn't contain any terminal node